

Uma Introdução à Teoria da Computação
(Versão Parcial: 11 de fevereiro de 2005)

Favor não distribuir

Michael Sipser

Traduzido do original em inglês
Introduction to the Theory of Computation
(PWS Publishing Company ©1997)
por Ruy J. Guerra B. de Queiroz

Índice

Prefácio	v
Ao(À) estudante	v
Ao(À) educador(a)	vi
A presente edição	vii
Realimentação para o autor	vii
Agradecimentos	vii
0 Introdução	1
0.1 Autômatos, computabilidade, e complexidade	1
Teoria da complexidade	1
Teoria da computabilidade	2
Teoria dos autômatos	3
0.2 Noções matemáticas e terminologia	3
Conjuntos	3
Seqüências e uplas	5
Funções e relações	6
Grafos	8
Cadeias e linguagens	10
Lógica booleana	11
Resumo de termos matemáticos	12
0.3 Definições, teoremas, e provas	13
Encontrando provas	13
0.4 Tipos de prova	16
Provas por construção	16
Prova por contradição	17
Prova por indução	18
<i>Exercícios e Problemas</i>	20
Parte Um: Autômatos e Linguagens	25
1 Linguagens Regulares	25
1.1 Autômatos finitos	25
Definição formal de um autômato finito	28
Exemplos de autômatos finitos	30
Definição formal de computação	32
Projetando autômatos finitos	34
As operações regulares	36

1.2	Não-determinismo	39
	Definição formal de um autômato finito não-determinístico	45
	Equivalência de AFN's e AFD's	46
	Fecho sob as operações regulares	50
1.3	Expressões regulares	54
	Definição formal de uma expressão regular	55
	Equivalência com autômatos finitos	57
1.4	Linguagens não-regulares	63
	O lema do bombeamento para linguagens regulares	64
	<i>Exercícios e Problemas</i>	68
2	Linguagens Livres-do-Contexto	83
2.1	Gramáticas livres-do-contexto	84
	Definição formal de uma gramática livre-do-contexto	85
	Projetando gramáticas livres-do-contexto	87
	Ambigüidade	88
	Forma normal de Chomsky	89
2.2	Autômatos a pilha	91
	Definição formal de um autômato a pilha	92
	Exemplos de autômatos a pilha	93
	Equivalência com gramáticas livres-do-contexto	95
2.3	Linguagens não-livres-do-contexto	101
	O lema do bombeamento para linguagens livres-do-contexto	101
	<i>Exercícios e Problemas</i>	105
Parte Dois: Teoria da Computabilidade		111
3	A Tese de Church-Turing	111
3.1	Máquinas de Turing	111
	Definição formal de uma máquina de Turing	113
	Exemplos de máquinas de Turing	115
3.2	Variantes de máquinas de Turing	119
	Máquinas de Turing multi-fitas	120
	Máquinas de Turing não-determinísticas	121
	Enumeradores	123
	Equivalência com outros modelos	124
3.3	A definição de algoritmo	125
	Os problemas de Hilbert	125
	Terminologia para descrever máquinas de Turing	127
	<i>Exercícios e Problemas</i>	129
4	Decidibilidade	133
4.1	Linguagens decidíveis	133
	Problemas decidíveis concernentes a linguagens regulares	134
	Problemas decidíveis concernentes a linguagens livres-do-contexto	137
4.2	O problema da parada	139
	O método da diagonalização	140
	O problema da parada é indecidível	144
	Uma linguagem Turing-irreconhecível	147

<i>Exercícios e Problemas</i>	148
5 Redutibilidade	151
5.1 Problemas indecidíveis da teoria de linguagens	152
Reduções via histórias de computação	155
5.2 Um problema indecidível simples	161
5.3 Redutibilidade por mapeamento	166
Funções computáveis	166
Definição formal de redutibilidade por mapeamento	167
<i>Exercícios e Problemas</i>	170
6 Tópicos Avançados em Computabilidade	173
6.1 O teorema da recursão	173
6.2 Decidibilidade de teorias lógicas	173
6.3 Redutibilidade de Turing	173
6.4 Uma definição de informação	173
Parte Três: Teoria da Complexidade	177
7 Complexidade de Tempo	177
7.1 Medindo complexidade	177
Notação O -grande e o -pequeno	178
Analizando algoritmos	180
Relacionamentos de complexidade entre modelos	183
7.2 A classe P	185
Tempo polinomial	185
Exemplos de problemas em P	187
7.3 A classe NP	192
Exemplos de problemas em NP	195
A questão P versus NP	197
7.4 NP-completude	197
Redutibilidade em tempo polinomial	198
Definição de NP-completude	201
O teorema de Cook–Levin	202
7.5 Problemas NP-completos adicionais	207
O problema da cobertura de vértices	208
O problema do caminho hamiltoniano	208
O problema da soma de subconjuntos	212
<i>Exercícios e Problemas</i>	214
8 Complexidade de Espaço	219
8.1 Teorema de Savitch	220
8.2 A classe PSPACE	220
8.3 PSPACE-completude	220
8.4 As classes L e NL	220
8.5 NL-completude	220
8.6 NL é igual a coNL	220

9 Intratabilidade	221
9.1 Teoremas de hierarquia	221
9.2 Relativização	224
9.3 Complexidade de circuito	224
10 Tópicos Avançados em Complexidade	225
10.1 Algoritmos de aproximação	225
10.2 Algoritmos probabilísticos	225
10.3 Alternação	225
10.4 Sistemas de prova interativa	225
10.5 Computação paralela	225
10.6 Criptografia	225
Bibliografia Selecionada	227
Índice Remissivo	230

Prefácio

Ao(À) estudante

Bem-vindo(a)!

Você está prestes a embarcar no estudo de um assunto fascinante e importante: a teoria da computação. Ela compreende as propriedades matemáticas fundamentais do hardware, do software, e das aplicações de computadores. Estudando esse assunto buscamos determinar o que pode e o que não pode ser computado, quão rapidamente, com quanto de memória, e sobre que tipo de modelo computacional. O assunto tem conexões óbvias com a prática da engenharia, e, como em muitas ciências, ele também tem aspectos puramente filosóficos.

Sei que muitos de vocês estão ansiosos para estudar este material mas alguns podem não estar aqui por sua própria escolha. Você pode querer obter um grau em ciência ou engenharia da computação, e um curso em teoria é requerido—Deus sabe por que. Afinal de contas, teoria não é coisa velha, enjoada, e, pior ainda, irrelevante?

Para ver que teoria não é nem coisa velha nem enjoada, mas ao contrário bastante comprehensível e até interessante, continue leia. Ciência da computação teórica de fato tem muitas idéias grandes e fascinantes, mas ela também tem muitos detalhes pequenos e às vezes tediosos que podem ser cansativos. Aprender qualquer assunto é trabalho árduo, mas fica mais fácil e mais divertido se o assunto é devidamente apresentado. Meu objetivo principal ao escrever este livro é expor você aos aspectos genuinamente excitantes da teoria da computação, sem se perder em detalhes cansativos. Claro que a única maneira de determinar se teoria lhe interessa é tentar aprendê-la.

Teoria é relevante para a prática. Ela provê ferramentas conceituais que os praticantes usam em engenharia da computação. Projetar uma nova linguagem de programação para uma aplicação especializada? O que você aprendeu sobre *gramáticas* neste curso vem bem a calhar. Lidar com busca por cadeias e casamento de padrões? Lembre-se de *autômatos finitos* e *expressões regulares*. Confrontado com um problema que parece requerer mais tempo de computador do que você pode suportar? Pense no que você aprendeu sobre *NP-completude*. Várias áreas de aplicação tais como protocolos criptográficos modernos, se sustentam em princípios teóricos que você vai aprender aqui.

Teoria também é relevante para você porque ela lhe mostra um lado mais simples, e mais elegante, dos computadores, os quais normalmente consideramos como sendo máquinas complicadas. Os melhores projetos e aplicações de computadores são concebidos com elegância em mente. Um curso teórico pode elevar seu sentido estético e ajudá-lo a construir sistemas mais bonitos.

Finalmente, teoria é bom para você porque estudá-la expande sua mente. Tecnologia de computadores muda rapidamente. Conhecimento técnico específico, embora útil

hoje, fica desatualizado em apenas uns poucos anos. Considere por outro lado as habilidades de pensar, exprimir-se claramente e precisamente, para resolver problemas, e saber quando você não resolveu um problema. Essas habilidades têm valor duradouro. Estudar teoria treina você nessas áreas.

Considerações práticas à parte, quase todo mundo trabalhando com computadores tem curiosidade sobre essas criações impressionantes, suas capacidades, e suas limitações. Um novo ramo da matemática cresceu nos últimos 30 anos para responder a certas questões básicas. Aqui está uma que permanece sem solução: Se eu lhe der um número grande, digamos, com 500 dígitos, você pode encontrar seus fatores (os números que o dividem), em uma quantidade de tempo razoável? Mesmo usando um supercomputador, ninguém atualmente conhece como fazer isso em todos os casos *sem o tempo de vida do universo!* O problema da fatoração está relacionado a certos códigos secretos em criptossistemas modernos. Encontre um maneira rápida de fatorar e a fama é toda sua!

Ao(À) educador(a)

Este livro pretende ser um texto para o final da graduação ou o início da pós-graduação em teoria da computação. Ele contém um tratamento do assunto desenhado em torno de teoremas e provas. Fiz algum esforço para acomodar estudantes com pouca experiência prévia em provar teoremas, embora estudantes mais experientes terão uma vida mais fácil.

Meu objetivo principal ao apresentar o material tem sido torná-lo claro e interessante. Ao fazer isso, tenho enfatizado a intuição e “a grande figura” no assunto em detrimento de alguns detalhes de mais baixo nível.

Por exemplo, muito embora apresente o método de prova por indução no Capítulo 0 juntamente com outros preliminares matemáticos, ela não desempenha um papel importante subsequentemente. Geralmente não apresento as provas por indução usuais da corretude de várias construções relativas a autômatos. Se apresentadas claramente, essas construções convencem e não necessitam de muito argumento. Uma indução pode confundir ao invés de esclarecer porque a própria indução é uma técnica um tanto sofisticada que muitos acham misteriosa. Trabalhar o óbvio com uma indução corre o risco de ensinar aos estudantes que prova matemática é uma manipulação formal em vez de ensiná-los o que é e o que não é um argumento cogente.

Um segundo exemplo ocorre nas Partes II e III, onde descrevo algoritmos em prosa ao invés de pseudocódigo. Não gasto muito tempo programando máquinas de Turing (ou quaisquer outros modelos formais). Estudantes hoje vêm com uma formação em programação e acham que a tese de Church–Turing é auto-evidente. Daí não apresento simulações excessivamente longas de um modelo por outro para estabelecer sua equivalência.

Além de dar intuição adicional e suprimir alguns detalhes, dou o que poderia ser chamado uma apresentação clássica do material. Muitos teóricos acharão a escolha do material, terminologia, e ordem de apresentação consistente com os de outros livros-texto largamente utilizados. Introduzí terminologia original em apenas uns poucos lugares, quando achei a terminologia padrão particularmente obscura ou confusa. Por exemplo introduzo o termo *redutibilidade por mapeamento* ao invés de *redutibilidade muitos-para-um*.

Prática por meio de resolução de problemas é essencial para aprender qualquer assunto matemático. Neste livro, os problemas são organizados em duas categorias

principais chamadas *Exercícios* e *Problemas*. Os Exercícios revisam definições e conceitos. Os Problemas requerem alguma engenhosidade. Problemas marcados com um asterisco são mais difíceis. Tentei tornar tanto os Exercícios quanto os Problemas desafios interessantes.

A presente edição

Introdução à Teoria da Computação primeiro apareceu como uma Edição Preliminar em capa mole. A presente edição difere da Edição Preliminar de várias maneiras substanciais. Os três últimos capítulos são novos: Capítulo 8 sobre complexidade de espaço; Capítulo 9 sobre intratabilidade demonstrável; e Capítulo 10 sobre tópicos avançados em teoria da complexidade. O Capítulo 6 foi expandido para incluir vários tópicos avançados em teoria da computabilidade. Outros capítulos foram melhorados através da inclusão de exemplos e exercícios adicionais.

Comentários de instrutores e estudantes que usaram a Edição Preliminar foram úteis para o polimento dos Capítulos 0 a 7. Obviamente, os erros que eles reportaram foram corrigidos nesta edição.

Os Capítulos 6 e 10 dão um apanhado de vários tópicos mais avançados em teorias de computabilidade e complexidade. Eles não se pretendem compreender uma unidade coesa da maneira que os capítulos remanescentes o são. Esses capítulos foram incluídos para permitir ao instrutor selecionar tópicos opcionais que podem ser de interesse do estudante mais exigente. Os próprios tópicos variam amplamente. Alguns, tais como *redutibilidade de Turing* e *alternação*, são extensões diretas de outros conceitos no livro. Outros, tais como *teorias lógicas decidíveis* e *criptografia*, são breves introduções a grandes áreas.

Realimentação para o autor

A internet provê novas oportunidades para interação entre autores e leitores. Tenho recebido bastante mensagens eletrônicas oferecendo sugestões, elogios, e críticas, e reportando erros na Edição Preliminar. Por favor continue a se corresponder! Tento responder a cada mensagem pessoalmente, quando o tempo permite. O endereço eletrônico para correspondência relacionada a este livro é

`sipserbook@math.mit.edu`

Uma página na internet que contém uma lista de erros é mantida no endereço abaixo. Outro material pode ser adicionado àquela página para ajudar a instrutores e estudantes. Diga-me o que gostaria de ver ali. O endereço é

`http://www-math.mit.edu/~sipser/book.html`

Agradecimentos

Eu não poderia ter escrito este livro sem a ajuda de muitos amigos, colegas, e minha família.

Quero agradecer aos professores que ajudaram a dar forma a meu ponto de vista científico e estilo educacional. Cinco deles se destacam. Meu orientador de tese, Manuel Blum, a quem devo uma nota especial por sua maneira única de inspirar estudantes através da clareza de pensamento, entusiasmo, e cuidado. Ele é um modelo para mim e

para muitos outros. Sou agradecido a Richard Karp por me introduzir à teoria da complexidade, a John Addison por me ensinar lógica e passar aqueles maravilhosos conjuntos de tarefas para casa, a Juris Hartmanis por me introduzir à teoria da computação, e a meu pai por me introduzir na matemática, computadores, e a arte de ensinar.

Este livro cresceu de notas de um curso que ensinei no MIT durante os últimos 15 anos. Estudantes em minhas turmas tomaram tais notas a partir de minhas aulas. Espero que eles me perdoem por não listá-los todos. Meus assistentes de ensino durante anos, Avrim Blum, Thang Bui, Andrew Chou, Benny Chor, Stavros Cosmadakis, Aditi Dhagat, Wayne Goddard, Parry Husbands, Dina Kravets, Jakov Kučan, Brian O'Neill, Ioana Popescu, e Alex Russell, me ajudaram a editar e expandir essas notas e me forneceram alguns dos problemas para tarefa de casa.

Quase três anos atrás, Tom Leighton me persuadiu a escrever um livro-texto sobre a teoria da computação. Eu vinha pensando em fazê-lo durante algum tempo, mas foi preciso a persuasão de Tom para fazer a teoria virar prática. Aprecio seus conselhos generosos sobre escrever livro e sobre muitas outras coisas.

Quero agradecer a Eric Bach, Peter Beebe, Cris Calude, Marek Chrobak, Anna Chefter, Guang-Ien Cheng, Elias Dahlhaus, Michael Fisher, Steve Fisk, Lance Fortnow, Henry J. Friedman, Jack Fu, Seymour Ginsbourg, Oded Goldreich, Brian Grossman, David Harel, Micha Hofri, Dung T. Huynh, Neil Jones, H. Chad Lane, Kevin Lin, Michael Loui, Silvio Micali, Tadao Murata, Christos Papadimitriou, Vaughan Pratt, Daniel Rosenband, Brian Scassellati, Ashish Sharma, Nir Shavit, Alexander Shen, Ilya Shlyakhter, Matt Stallman, Perry Susskind, Y. C. Tay, Joseph Traub, Osamu Watanabe, Peter Widemayer, David Williamson, Derick Wood, e Charles Yang pelos comentários, sugestões, e assistência à medida que a escrita progrediu.

As seguintes pessoas acrescentaram comentários que melhoraram este livro: Isam M. Abdelhameed, Eric Allender, Michelle Atherton, Rolfe Blodgett, Al Briggs, Brian E. Brooks, Jonathan Buss, Jin Yi Cai, Steve Chapel, David Chow, Michael Ehrlich, Yaakov Eisenberg, Farzan Fallah, Shaun Flisakowski, Hjalmyr Hafsteinsson, C. R. Hale, Maurice Herlihy, Vegard Holmedahl, Sandy Irani, Kevin Jiang, Rhys Price Jones, James M. Jowdy, David M. Martin Jr., Manrique Mata-Montero, Ryota Matsuura, Thomas Minka, Farooq Mohammed, Tadao Murata, Jason Murray, Hideo Nagashashi, Kazuo Ohta, Constantine Papageorgiou, Joseph Raj, Rick Regan, Rhonda A. Reumann, Michael Rintzler, Arnold L. Rozenberg, Larry Roske, Max Rozenoer, Walter L. Ruzzo, Sanathan Sahgal, Leonard Schulman, Steve Seiden, Joel Seiferas, Ambuj Singh, David J. Stucki, Jayram S. Thathachar, H. Venkateswaran, Tom Whaley, Christopher Van Wyk, Kyle Young, and Kyoung Hwan Yun.

Robert Sloan usou uma versão anterior do manuscrito deste livro em uma turma que ele ensinou e me passou inestimáveis comentários e idéias de sua experiência com o manuscrito. Mark Herschberg, Kazuo Ohta, e Latanya Sweeney leram partes do manuscrito e sugeriram melhoramentos extensos. Shafi Goldwasser me ajudou com o material do Capítulo 10.

Recebí suporte técnico especialista de William Baxter da Superscript, que escreveu o pacote de macros \LaTeX que implementa o desenho interior, e de Larry Nolan do departamento de matemática do MIT, que mantém tudo funcionando.

Tem sido um prazer trabalhar com o pessoal da PWS Publishing na criação do produto final. Mencione Michael Sugarman, David Dietz, Elise Kaiser, Monique Calello, Susan Garland e Tanja Brull porque tive maior contato com eles, mas sei que muitos outros estiveram envolvidos também. Obrigado a Jerry Moore pela edição de cópia (*copy editing*), a Diane Levy pelo projeto da capa, e a Catherine Hawkes pelo desenho do interior.

Agradeço à National Science Foundation pelo apoio fornecido sob o *grant* CCR-9503322.

Meu pai, Kenneth Sipser, e irmã, Laura Sipser, converteram os diagramas do livro para forma eletrônica. Minha outra irmã, Karen Fisch, nos salvou em várias emergências computacionais, e minha mãe, Justine Sipser, ajudou com os conselhos maternos. Agradeço-os por contribuir sob circunstâncias difíceis, incluindo prazos insanos e software recalcitrante.

Finalmente, meu amor vai para minha esposa, Ina, e minha filha, Rachel. Obrigado por aguentar tudo isso.

*Cambridge, Massachusetts
Outubro de 1996*

Michael Sipser

Capítulo 0

Introdução

Vamos começar com uma visão geral daquelas áreas da teoria da computação que apresentamos neste curso. Então, você terá uma chance de aprender e/ou revisar alguns conceitos matemáticos que você vai precisar mais adiante.

0.1 Autômatos, computabilidade, e complexidade

Este livro enfoca três áreas tradicionalmente centrais da teoria da computação: autômatos, computabilidade, e complexidade. Elas são interligadas pela questão:

Quais são as capacidades e limitações fundamentais dos computadores?

Essa questão vai lá atrás nos anos 1930's quando lógicos matemáticos primeiramente começaram a explorar o significado de computação. Avanços tecnológicos desde aquele tempo tem aumentado enormemente nossa capacidade de computar e tem trazido essa questão do domínio da teoria para o mundo do interesse prático.

Em cada uma das três áreas—autômatos, computabilidade, e complexidade—essa questão é interpretada diferentemente, e as respostas variam conforme a interpretação. Após este capítulo introdutório, exploraremos cada área em uma parte separada deste livro. Aqui, introduzimos essas partes em ordem reversa para o iniciante.

Teoria da complexidade

Problemas computacionais vêm em diferentes variedades: alguns são fáceis e alguns difíceis. Por exemplo, o problema da ordenação é um fácil. Digamos que você precise de arranjar uma lista de números em ordem ascendente. Mesmo um pequeno computador pode ordenar um milhão de números bastante rapidamente. Compare isso a um problema de escalonamento. Digamos que você tenha que encontrar um escalonamento de classes para a universidade inteira para satisfazer algumas restrições razoáveis, tal como nunca duas aulas têm lugar na mesma sala no mesmo tempo. O problema do escalonamento parece ser muito mais difícil que o problema da ordenação. Se você tem somente mil aulas, encontrar o melhor escalonamento pode requerer séculos, até mesmo com um supercomputador.

O que faz alguns problemas computacionalmente difíceis e outros fáceis?

Essa é a questão central da teoria da complexidade. Incrivelmente, não sabemos a resposta a ela, embora ela tenha sido intensamente pesquisada durante os últimos 25 anos. Mais adiante, exploraremos essa fascinante questão e algumas de suas ramificações.

Em uma das importantes conquistas da teoria da complexidade até agora, pesquisadores descobriram um elegante esquema para classificar problemas conforme sua dificuldade computacional. Ele é análogo à tabela periódica para classificar elementos conforme suas propriedades químicas. Usando esse esquema, podemos demonstrar um método para dar evidência que certos problemas são computacionalmente difíceis, ainda que somos incapazes de provar que eles o são.

Você tem várias opções quando você se confronta com um problema que parece ser computacionalmente difícil. Primeiro, entendendo qual aspecto do problema é a raiz da dificuldade, você pode ser capaz de alterá-lo de modo que o problema seja mais solúvel. Segundo, você pode ser capaz de se contentar com menos que uma solução perfeita para o problema. Em certos casos encontrar soluções que apenas aproximam a perfeita é relativamente fácil. Terceiro, alguns problemas são difíceis somente na situação do pior caso, mas fácil na maior parte do tempo. Dependendo da aplicação, você pode ficar satisfeito com um procedimento que ocasionalmente é lento mas usualmente roda rapidamente. Finalmente, você pode considerar tipos alternativos de computação, tais como computação aleatorizada, que pode acelerar certas tarefas.

Uma área aplicada que tem sido afetada diretamente pela teoria da complexidade é o velho campo da criptografia. Na maioria dos campos, um problema computacional fácil é preferível a um difícil porque os fáceis são mais baratos para resolver. Criptografia é incomum porque ela especificamente requer problemas computacionais que sejam difíceis, ao invés de fáceis, porque códigos secretos devem ser tratados para serem difíceis de quebrar sem a chave secreta ou senha. A teoria da complexidade tem levado os criptógrafos na direção de problemas computacionalmente difíceis em torno dos quais eles têm projetado novos códigos revolucionários.

Teoria da computabilidade

Durante a primeira metade do século XX, matemáticos como Kurt Gödel, Alan Turing, e Alonzo Church descobriram que certos problemas básicos não podem ser resolvidos por computadores. Um exemplo desse fenômeno é o problema de se determinar se um enunciado matemático é verdadeiro ou falso. Essa tarefa é o pão com manteiga dos matemáticos. Parece uma questão natural para resolução por computador porque ela reside estritamente dentro do domínio da matemática. Mas nenhum algoritmo de computador pode realizar essa tarefa.

Entre as consequências desse resultado profundo estava o desenvolvimento de idéias concernentes aos modelos teóricos de computadores que em algum momento ajudariam a levar à construção de computadores reais.

As teorias da computabilidade e da complexidade são intimamente relacionadas. Na teoria da complexidade, o objetivo é classificar problemas como fáceis e difíceis, enquanto que na teoria da computabilidade a classificação de problemas é por meio da separação entre os que são solúveis e os que não o são. A teoria da computabilidade introduz vários dos conceitos usados na teoria da complexidade.

Teoria dos autômatos

Teoria dos autômatos lida com as definições e propriedades de modelos de computação. Esses modelos desempenham um papel em diversas áreas aplicadas da ciência da computação. Um modelo, chamado *autômato finito*, é usado para processamento de texto, compiladores, e projeto de hardware. Um outro modelo, chamado *gramática livre-do-contexto*, é usado em linguagens de programação e inteligência artificial.

Teoria dos autômatos é um excelente lugar para começar a estudar a teoria da computação. As teorias de computabilidade e complexidade requerem uma definição precisa de um *computador*. A teoria dos autômatos permite praticar com definições formais de computação à medida que ela introduz conceitos relevantes a outras áreas não-teóricas da ciência da computação.

0.2 Noções matemáticas e terminologia

Como em qualquer assunto matemático, começamos com uma discussão dos objetos matemáticos básicos, ferramentas, e notação que esperamos usar.

Conjuntos

Um **conjunto** é um grupo de objetos representado como uma unidade. Conjuntos podem conter qualquer tipo de objeto, incluindo números, símbolos, e até mesmo outros conjuntos. Os objetos em um conjunto são chamados seus **elementos** ou **membros**. Conjuntos podem ser descritos formalmente de várias maneiras. Uma maneira é listar seus elementos dentro de chaves. Por conseguinte o conjunto

$$\{7, 21, 57\}$$

contém os elementos 7, 21, e 57. Os símbolos \in e \notin denotam pertinência e não-pertinência, respectivamente. Escrevemos $7 \in \{7, 21, 57\}$ e $8 \notin \{7, 21, 57\}$. Para dois conjuntos A e B , dizemos que A é um **subconjunto** de B , escrito $A \subseteq B$, se todo membro de A também é um membro de B . Dizemos que A é um **subconjunto próprio** de B , escrito $A \subset B$, se A é um subconjunto de B e não é igual a B .

A ordem de descrição de um conjunto não importa, nem também a repetição de seus membros. Obtemos o mesmo conjunto escrevendo $\{57, 7, 7, 7, 21\}$. Se quisermos levar o número de ocorrências de membros em consideração chamamos o grupo de um **multiconjunto** ao invés de um conjunto. Por conseguinte $\{7\}$ e $\{7, 7\}$ são diferentes enquanto multiconjuntos mas idênticos enquanto conjuntos. Um **conjunto infinito** contém uma quantidade infinita de elementos. Não podemos escrever uma lista de todos os elementos de um conjunto infinito, portanto às vezes usamos a notação “...” para indicar “continue a seqüência para sempre.” Por conseguinte escrevemos o conjunto de **números naturais** \mathbb{N} como

$$\{1, 2, 3, \dots\}.$$

O conjunto de **inteiros** \mathbb{Z} é escrito

$$\{\dots, -2, -1, 0, 1, 2, \dots\}.$$

O conjunto com 0 membros é chamado o **conjunto vazio** e é escrito \emptyset .

Quando desejamos descrever um conjunto contendo elementos conforme alguma regra, escrevemos $\{n \mid \text{regra sobre } n\}$. Por conseguinte, $\{n \mid n = m^2 \text{ para algum } n \in \mathbb{N}\}$ significa o conjunto de quadrados perfeitos.

Se temos dois conjuntos A e B , a **união** de A e B , escrita $A \cup B$, é o conjunto que obtemos combinando todos os elementos em A e B em um único conjunto. O **complemento** de A , escrito \bar{A} , é o conjunto de todos os elementos sob consideração que *não* estão em A .

Como é frequentemente o caso em matemática, uma figura visual ajuda a esclarecer um conceito. Para conjuntos, usamos um tipo de figura chamado **diagrama de Venn**. Ele representa conjuntos como regiões delimitadas por linhas circulares. Seja o conjunto *INICIA-t* como sendo o conjunto de todas as palavras em português que começam com a letra “t.” Por exemplo, na Figura 1 o círculo representa o conjunto *INICIA-t*. Vários membros desse conjunto são representados como pontos dentro do círculo.

Figura 1: Diagrama de Venn para o conjunto de palavras em português começando com “t”

Igualmente, representamos o conjunto *TERMINA-z* de palavras em português que terminam com “z” na Figura 2.

Figura 2: Diagrama de Venn para o conjunto de palavras em português terminando com “z”

Para representar ambos os lados no mesmo diagrama de Venn temos que desenhá-los de modo que eles se sobreponham, indicando que eles compartilham alguns elementos, como mostrado na Figura 3. Por exemplo, a palavra *tenaz* está em ambos os conjuntos. A figura também contém um círculo para o conjunto *INICIA-j*. Ele não sobrepõe-se ao círculo para *INICIA-t* porque nenhuma palavra reside em ambos os conjuntos.

Figura 3: Círculos que se sobrepõem indicam elementos em comum

O diagramas de Venn da Figura 4 mostram a união e a interseção dos conjuntos A e B .

Figura 4: Diagramas para (a) $A \cup B$ e (b) $A \cap B$

Seqüências e uplas

Uma *seqüência* de objetos é uma lista desses objetos em alguma ordem. Usualmente designamos uma seqüência escrevendo a lista dentro de parênteses. Por exemplo, a seqüência 7, 21, 57 seria escrita

$$(7, 21, 57).$$

Em um conjunto a ordem não importa, mas em uma seqüência sim. Daí (7, 21, 57) não é o mesmo que (57, 7, 21). Repetição não é permitido em um conjunto mas é permitido em uma seqüência, portanto (7, 7, 21, 57) é diferente de ambas as outras seqüências, enquanto que o conjunto {7, 21, 57} é idêntico ao conjunto {7, 7, 21, 57}.

Como com conjuntos, seqüências podem ser finitas ou infinitas. Seqüências finitas são frequentemente chamadas *uplas*. Uma seqüência com k elementos é uma k -*upla*. Por conseguinte, (7, 21, 57) é uma 3-upla. Uma 2-upla é também chamada um *par*.

Conjuntos e seqüências podem aparecer como elementos de outros conjuntos e seqüências. Por exemplo, o *conjunto das partes* de A é o conjunto de todos os subconjuntos de A . Se A é o conjunto {0, 1}, o conjunto das partes de A é o conjunto $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$. O conjunto de todos os pares cujos elementos são 0's e 1's é {(0, 0), (0, 1), (1, 0), (1, 1)}.

Se A e B são dois conjuntos, o *produto cartesiano* ou *produto cruzado* de A e B , escrito $A \times B$, é o conjunto de todos os pares nos quais o primeiro elemento é um membro de A e o segundo elemento é um membro de B .

Exemplo 0.1

Se $A = \{1, 2\}$ e $B = \{x, y, z\}$,

$$A \times B = \{(1, x), (1, y), (1, z), (2, x), (2, y), (2, z)\}.$$

Podemos também tomar o produto cartesiano de k conjuntos, A_1, A_2, \dots, A_k , escrito $A_1 \times A_2 \times \dots \times A_k$. É o conjunto consistindo de todas as k -uplas (a_1, a_2, \dots, a_k) onde $a_i \in A_i$.

Exemplo 0.2

Se A e B são como no Exemplo 0.1,

$$A \times B \times A = \{(1, x, 1), (1, x, 2), (1, y, 1), (1, y, 2), (1, z, 1), (1, z, 2), (2, x, 1), (2, x, 2), (2, y, 1), (2, y, 2), (2, z, 1), (2, z, 2)\}.$$

Se temos o produto cartesiano de um conjunto consigo mesmo, usamos a notação abreviada

$$\overbrace{A \times A \times \dots \times A}^k = A^k.$$

Exemplo 0.3

O conjunto \mathbb{N}^2 é igual a $\mathbb{N} \times \mathbb{N}$. Consiste de todos os pares de números naturais. Também podemos escrevê-lo como $\{(i, j) \mid i, j \geq 1\}$.

Funções e relações

Funções são centrais para a matemática. Uma *função* é um objeto que estabelece um relacionamento entrada-saída. Uma função toma uma entrada e produz uma saída. Em toda função, a mesma entrada sempre produz a mesma saída. Se f é uma função cujo valor de saída é b quando o valor de entrada é a , escrevemos

$$f(a) = b.$$

Uma função também é chamada um *mapeamento*, e, se $f(a) = b$, dizemos que f mapeia a em b .

Por exemplo, a função de valor absoluto abs toma um número x como entrada e retorna x se x é positivo e $-x$ se x é negativo. Por conseguinte $abs(2) = abs(-2) = 2$. Adição é um outro exemplo de uma função, escrita *adicao*. A entrada para a função adição é um par de números e a saída é a soma daqueles números.

O conjunto de possíveis entradas para a função é chamada seu *domínio*. As saídas de uma função vêm de um conjunto chamado seu *contradomínio*. A notação para dizer que f é uma função com domínio D e contradomínio R é

$$f : D \longrightarrow R.$$

No caso da função abs , se estamos trabalhando com inteiros, o domínio e o contradomínio são \mathbb{Z} , portanto escrevemos $abs : \mathbb{Z} \longrightarrow \mathbb{Z}$. No caso da função adição para inteiros, o domínio é o conjunto de pares de inteiros $\mathbb{Z} \times \mathbb{Z}$ e o contradomínio é \mathbb{Z} , portanto escrevemos $adicao : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$. Note que uma função pode não necessariamente usar todos os elementos do contradomínio especificado. A função nunca toma o valor -1 muito embora $-1 \in \mathbb{Z}$. Uma função que realmente usa todos os elementos do contradomínio é dita ser *sobre* o contradomínio.

Podemos descrever uma função específica de várias maneiras. Uma maneira é com um procedimento para computar uma saída a partir de uma entrada especificada. Uma outra maneira é com uma tabela que lista todas as possíveis entradas e dá a saída para cada entrada.

Exemplo 0.4

Considere a função $f : \{0, 1, 2, 3, 4\} \longrightarrow \{0, 1, 2, 3, 4\}$.

n	$f(n)$
0	1
1	2
2	3
3	4
4	0

Essa função adiciona 1 a sua entrada e então dá como saída o resultado módulo 5. Um número módulo m é o resto da divisão por m . Por exemplo, o ponteiro de minutos no mostrador de um relógio conta módulo 60. Quando fazemos aritmética modular definimos $\mathbb{Z}_m = \{0, 1, 2, \dots, m - 1\}$. Com essa notação, a função supramencionada f tem a forma $f : \mathbb{Z}_5 \longrightarrow \mathbb{Z}_5$.

Exemplo 0.5

0.2. NOÇÕES MATEMÁTICAS E TERMINOLOGIA 7

Às vezes uma tabela bidimensional é usada se o domínio da função é o produto cartesiano de dois conjuntos. Aqui vai uma outra função $g : \mathcal{Z}_4 \times \mathcal{Z}_4 \rightarrow \mathcal{Z}_4$. A entrada na linha rotulada i e a coluna rotulada j na tabela é o valor de $g(i, j)$.

g	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

A função g é a função adição módulo 4.

Quando o domínio de uma função f é $A_1 \times \cdots \times A_k$ para alguns conjuntos A_1, \dots, A_k , a entrada para f é uma k -upla (a_1, a_2, \dots, a_k) e chamamos os a_i 's de **argumentos** para f . Uma função com k argumentos é chamada uma **função k -ária** e k é chamada a **aridade** da função. Se k é 1, f é uma **função unária**. Se k é 2, f é uma **função binária**. Certas funções binárias familiares são escritas numa **notação infixa** especial, com o símbolo para a função colocado entre seus dois argumentos, ao invés de em **notação prefixada**, com o símbolo precedendo os argumentos. Por exemplo, a função adição *adicao* usualmente é escrita em notação infixa com o símbolo + entre seus dois argumentos como em $a + b$ ao invés de em notação prefixada *adicao*(a, b).

Um **predicado** ou **propriedade** é uma função cujo contradomínio é {VERDADEIRO, FALSO}. Por exemplo, seja *par* uma propriedade que é VERDADEIRO se sua entrada é um número par e FALSO se sua entrada é um número ímpar. Por conseguinte $\text{par}(4) = \text{VERDADEIRO}$ e $\text{par}(5) = \text{FALSO}$.

Um propriedade cujo domínio é um conjunto de k -uplas $A \times \cdots \times A$ é chamada uma **relação**, uma **relação k -ária**, ou uma **relação k -ária sobre A** . Um caso comum é uma relação 2-ária chamada uma **relação binária**. Quando escrevendo uma expressão envolvendo uma relação binária, comumente usamos notação infixa. Por exemplo, “menor que” é uma relação usualmente escrita com o símbolo de operação infixo $<$. “Igualdade,” escrito com o símbolo $=$ é uma outra relação familiar. Se R é uma relação binária, o enunciado aRb significa que $aRb = \text{VERDADEIRO}$. Igualmente se R é uma relação k -ária, o enunciado $R(a_1, \dots, a_k)$ significa que $R(a_1, \dots, a_k) = \text{VERDADEIRO}$.

Exemplo 0.6

Num jogo infantil chamado Tesoura-Papel-Pedra, os dois jogadores simultaneamente selecionam um membro do conjunto {TESOURA, PAPEL, PEDRA} e indicam suas escolhas com sinais feitos com a mão. Se as duas escolhas são a mesma, o jogo começa novamente. Se as escolhas diferem, um jogador vence, conforme a relação *bate*.

<i>bate</i>	TESOURA	PAPEL	PEDRA
TESOURA	FALSO	VERDADEIRO	FALSO
PAPEL	FALSO	FALSO	VERDADEIRO
PEDRA	VERDADEIRO	FALSO	FALSO

Dessa tabela determinamos que TESOURA *bate* PAPEL é VERDADEIRO e que PAPEL *bate* TESOURA é FALSO.

Às vezes descrever predicados com conjuntos ao invés de funções é mais conveniente. O predicado $P : D \rightarrow \{\text{VERDADEIRO}, \text{FALSO}\}$ pode ser escrito (D, S) , onde $S = \{a \in D \mid P(a) = \text{VERDADEIRO}\}$, ou simplesmente S se o domínio D é

óbvio do contexto. Portanto a relação *bate* pode ser escrita

$$\{(TESOURA, PAPEL), (PAPEL, PEDRA), (PEDRA, TESOURA)\}.$$

Um tipo especial de relação binária, chamada uma *relação de equivalência* captura a noção de dois objetos sendo iguais em alguma característica. Uma relação binária R é uma relação de equivalência se R satisfaz três condições:

1. R é **reflexiva** se para todo x , xRx ;
2. R é **simétrica** se para todo x e todo y , xRy se e somente se yRx ; e
3. R é **transitiva** se para todo x , y , e z , xRy e yRz implica xRz .

Exemplo 0.7

Defina uma relação de equivalência sobre os números naturais, escrita \equiv_7 . Para $i, y \in \mathbb{N}$ digamos que $i \equiv_7 j$, se $i - j$ é um múltiplo de 7. Essa é uma relação de equivalência porque ela satisfaz as três condições. Primeiro, ela é reflexiva, pois $i - i = 0$, que é um múltiplo de 7. Segundo, ela é simétrica, pois $i - j$ é um múltiplo de 7 se $j - i$ é um múltiplo de 7. Terceiro, ela é transitiva, pois sempre que $i - j$ é um múltiplo de 7 e $j - k$ é um múltiplo de 7, então $i - k = (i - j) + (j - k)$ é a soma de dois múltiplos de 7 e portanto também é um múltiplo de 7.

Grafos

Um **grafo não-direcionado**, ou simplesmente um **grafo**, é um conjunto de pontos com linhas conectando alguns dos pontos. Os pontos são chamados **nós** ou **vértices**, e as linhas são chamadas **arestas**, como mostrado na Figura 5.

Figura 5: Exemplos de grafos

O número de arestas em nó específico é o **grau** daquele nó. Na Figura 5(a) todos os nós têm grau 2. Na Figura 5(b) todos os nós têm grau 3. Não mais que uma aresta é permitida entre quaisquer dois nós.

Em um grafo G que contém os nós i e j , o par (i, j) representa a aresta que conecta i e j . A ordem de i e j não importa em um grafo não-direcionado, portanto os pares (i, j) e (j, i) representam a mesma aresta. Às vezes descrevemos arestas com conjuntos, como em $\{i, j\}$, ao invés de pares porque a ordem dos nós não é importante. Se V é o conjunto de nós de G e E é o conjunto de arestas, dizemos que $G = (V, E)$. Podemos descrever um grafo com um diagrama ou mais formalmente especificando V e E . Por exemplo, uma descrição formal do grafo na Figura 5(a) é

$$(\{1, 2, 3, 4, 5\}, \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 1)\}),$$

e uma descrição formal do grafo na Figura 5(b) é

$$(\{1, 2, 3, 4\}, \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}),$$

Grafos frequentemente são usados para representar dados. Nós podem ser cidades e arestas as estradas conectando-as, ou nós podem ser componentes elétricos e arestas

os fios entre eles. Às vezes, por conveniência, rotulamos os nós e/ou as arestas de um grafo, que então é chamado um **grafo rotulado**. A Figura 6 mostra um grafo cujos nós são cidades e cujas arestas são rotuladas com o custo em reais da tarifa aérea sem escalas mais barata para viajar entre aquelas cidades se voar sem escalas entre elas é possível.

Figura 6: Tarifas aéreas sem-escalas mais baratas entre várias cidades

Dizemos que um grafo G é um **subgrafo** do grafo H se os nós de G são um subconjunto dos nós de H . Como mostrado na Figura 7, as arestas de G são as arestas de H nos nós correspondentes.

Figura 7: O grafo G (mais escuro) é um subgrafo de H

Um **caminho** em um grafo é uma seqüência de nós conectados por arestas. Um **caminho simples** é um caminho que não repete nenhum nó. Um grafo é **conexo** se cada dois nós têm um caminho entre eles. Um caminho é um **ciclo** se ele começa e termina no mesmo nó. Um **ciclo simples** é um ciclo de comprimento 3 ou mais que não repete nenhum nó exceto o primeiro e o último. Um grafo é uma **árvore** se ele é conexo e não tem ciclos simples, conforme mostrado na Figura 8. Os nós de grau 1 em uma árvore são chamados as **folhas** da árvore. Às vezes existe um nó especialmente designado de uma árvore chamado de **raiz**.

Figura 8: (a) Um caminho em um grafo, (b) um ciclo em um grafo, e (c) uma árvore

Se ele tem setas ao invés de linhas, o grafo é um **grafo não-direcionado**, como mostrado na Figura 9. O número de setas apontando de um nó específico é o **grau de saída** daquele nó, e o número de setas apontando para um nó específico é o **grau de entrada**.

Em um grafo direcionado representamos uma aresta de i para j como um par (i, j) . A descrição formal de um grafo direcionado G é (V, E) onde V é o conjunto de nós e E é o conjunto de arestas. A descrição formal do grafo na Figura 9 é

$$(\{1, 2, 3, 4, 5, 6\}, \{(1, 2), (1, 5), (2, 1), (2, 4), (5, 4), (5, 6), (6, 1), (6, 3)\}).$$

Um caminho no qual todas as setas apontam na mesma direção como seus passos é chamado um **caminho direcionado**. Um grafo direcionado é **fortemente conexo** se um caminho direcionado conecta todo par de nós.

Figura 9: Um grafo direcionado

Figura 10: O grafo de uma relação

Exemplo 0.8

O grafo direcionado representando a relação dada no Exemplo 6 é

Grafos direcionados se constituem numa ferramenta útil para visualizar relações binárias. Se R é uma relação binária cujo domínio é $D \times D$, um grafo rotulado $G = (D, E)$ representa R , onde $E = \{(x, y) \mid xRy\}$. O Exemplo 0.8 é uma ilustração.

Cadeias e linguagens

Cadeias de caracteres são blocos básicos fundamentais em ciência da computação. O alfabeto sobre o qual as cadeias são definidas pode variar com a aplicação. Para nossos propósitos, definimos um **alfabeto** como sendo qualquer conjunto finito. Os membros do alfabeto são os símbolos do alfabeto. Geralmente usamos letras gregas maiúsculas Σ e Γ para designar alfabetos e um formato no estilo de máquinas de datilografia para símbolos de um alfabeto. Os seguintes são uns poucos exemplos de alfabetos.

$$\Sigma_1 = \{0, 1\}$$

$$\Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

$$\Gamma = \{0, 1, x, y, z\}$$

Uma **cadeia sobre um alfabeto** é uma seqüência de símbolos daquele alfabeto, usualmente escritos um após o outro e não separados por vírgulas. Se $\Sigma_1 = \{0, 1\}$, então 01001 é uma cadeia sobre Σ_1 . Se $\Sigma_2 = \{a, b, c, \dots, z\}$, então abracadabra é uma cadeia sobre Σ_2 . Se w é uma cadeia sobre Σ , o **comprimento** de w , escrito $|w|$, é o número de símbolos que ela contém. A cadeia de comprimento zero é chamada a **cadeia vazia** e é escrita ε . A cadeia vazia desempenha o papel do 0 em um sistema numérico. Se w tem comprimento n , podemos escrever $w = w_1 w_2 \cdots w_n$ onde cada $w_i \in \Sigma$. O **reverso** de w , escrito w^R , é a cadeia obtida escrevendo w na ordem oposta (i.e., $w_n w_{n-1} \cdots w_1$). A cadeia z é uma **subcadeia** de w se z aparece consecutivamente em w . Por exemplo, cad é uma subcadeia de abracadabra.

Se temos a cadeia x de comprimento m e a cadeia y de comprimento n , a **concatenação** de x e y , escrita xy , é a cadeia obtida por justaposição de y ao final de x , como em $x_1 \cdots x_m y_1 \cdots y_n$. Para concatenar uma cadeia consigo mesma muitas vezes usamos a notação:

$$\overbrace{xx \cdots x}^k = x^k.$$

A **ordenação lexicográfica** de cadeias é a mesma que a ordenação familiar do dicionário, exceto que cadeias mais curtas precedem cadeias mais longas. Portanto a ordenação lexicográfica de todas as cadeias sobre o alfabeto $\{0, 1\}$ é $(\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots)$.

Uma **linguagem** é um conjunto de cadeias.

Lógica booleana

Lógica booleana é um sistema matemático construído em torno de dois valores VERDADEIRO e FALSO. Embora originalmente concebido como matemática pura, esse sistema é hoje considerado como sendo os fundamentos da eletrônica digital e do projeto de computadores. Os valores VERDADEIRO e FALSO são chamados *valores booleanos* e são frequentemente representados pelos valores 0 e 1. Usamos valores booleanos em situações com duas possibilidades, tais como um fio que pode ter uma tensão alta ou baixa, uma proposição que pode ser verdadeira ou falsa, ou uma questão que pode ser respondida sim ou não.

Podemos manipular valores booleanos com operações especialmente projetadas, chamadas *operações booleanas*. A operação booleana mais simples é a *negação* ou a operação NÃO, designada com o símbolo \neg . A negação de um valor booleano é o valor oposto. Por conseguinte $\neg 0 = 1$ e $\neg 1 = 0$. A *conjunção*, ou a operação E é designada com o símbolo \wedge . A conjunção de dois valores booleanos é 1 se ambos aqueles valores são 1. A *disjunção* ou a operação OU é designada com o símbolo \vee . A disjunção de dois valores booleanos é 1 se um daqueles valores é 1. Resumimos essa informação na seguinte tabela:

$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$\neg 0 = 1$
$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$\neg 1 = 0$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	
$1 \wedge 1 = 1$	$1 \vee 1 = 1$	

Usamos operações booleanas para combinar enunciados simples em expressões booleanas mais complexas, tal como usamos as operações aritméticas $+$ e \times para construir expressões aritméticas complexas. Por exemplo, se P é o valor booleano representando a veracidade do enunciado “o sol está brilhando” e Q representa a veracidade do enunciado “hoje é Segunda-Feira”, pode escrever $P \wedge Q$ para representar o valor verdade do enunciado “o sol está brilhando e hoje é Segunda-Feira” e igualmente $P \vee Q$ com *e* substituído por *ou*. Os valores P e Q são chamados os *operandos* da operação.

Várias outras operações booleanas ocasionalmente aparecem. O *ou exclusivo*, ou operação XOR é designada com o símbolo \oplus e é 1 se um dos mas não dois operandos são 1. A operação *igualdade*, escrita com o símbolo \leftrightarrow , é 1 se ambos os seus operandos têm o mesmo valor. Finalmente, a operação da *implicação* é designada com o símbolo \rightarrow e é 0 se seu primeiro operando é 1 e o seu segundo operando é 0; caso contrário \rightarrow é 1. Resumimos essa informação na seguinte tabela:

$0 \oplus 0 = 0$	$0 \leftrightarrow 0 = 1$	$0 \rightarrow 0 = 1$
$0 \oplus 1 = 1$	$0 \leftrightarrow 1 = 0$	$0 \rightarrow 1 = 1$
$1 \oplus 0 = 1$	$1 \leftrightarrow 0 = 0$	$1 \rightarrow 0 = 0$
$1 \oplus 1 = 0$	$1 \leftrightarrow 1 = 1$	$1 \rightarrow 1 = 1$

Podemos estabelecer vários relacionamentos entre essas operações. Na verdade, podemos expressar todas as expressões booleanas em termos das operações E e NÃO, como as identidades abaixo mostram. As duas expressões em cada linha são equivalentes. Cada linha expressa a operação na coluna da esquerda em termos de operações acima dela e E e NÃO.

$P \vee Q$	$\neg(\neg P \wedge \neg Q)$
$P \rightarrow Q$	$\neg P \vee Q$
$P \leftrightarrow Q$	$(P \rightarrow Q) \wedge (Q \rightarrow P)$
$P \oplus Q$	$\neg(P \leftrightarrow Q)$

A **lei distributiva** para E e OU são úteis na manipulação de expressões booleanas. Ela é semelhante à lei distributiva para adição e multiplicação, que enuncia que $a \times (b + c) = (a \times b) + (a \times c)$. A versão booleana vem nas duas seguintes formas:

- $P \wedge (Q \vee R)$ é igual a $(P \wedge Q) \vee (P \wedge R)$, e sua dual
- $P \vee (Q \wedge R)$ é igual a $(P \vee Q) \wedge (P \vee R)$.

Note que a dual da lei distributiva para adição e multiplicação não se verifica em geral.

Resumo de termos matemáticos

Alfabeto	Um conjunto finito de objetos chamados símbolos
Aresta	Uma linha em um grafo
Argumento	Uma entrada para uma função
Árvore	Um grafo conexo sem ciclos simples
Cadeia	Uma lista finita de símbolos a partir de um objeto
Cadeia vazia	A cadeia de comprimento zero
Caminho	Uma seqüência de nós em um grafo conectado por arestas
Caminho simples	Um caminho sem repetição
Ciclo	Um caminho que começa e termina no mesmo nó
Complemento	Uma operação sobre um conjunto, formando o conjunto de todos os elementos não presentes
Concatenação	Uma operação que junta cadeias de um conjunto com cadeias de um outro conjunto
Conjunção	operação booleana E
Conjunto	Um grupo de objetos
Conjunto vazio	Um conjunto sem qualquer membro
Contradomínio	O conjunto a partir do qual as saídas de uma função são retiradas
Disjunção	operação booleana OU
Domínio	O conjunto de possíveis entradas para uma função
Elemento	Um objeto em um conjunto
Função	Uma operação que traduz entradas em saídas
Grafo	Uma coleção de pontos e linhas conectando alguns pares de pontos
Grafo conexo	Um grafo com caminhos conectando cada dois nós
Grafo direcionado	Uma coleção de pontos e setas conectando alguns pares de pontos
Interseção	Uma operação sobre conjuntos formando o conjunto de elementos comuns
<i>k</i> -upla	Uma lista de <i>k</i> objetos
Linguagem	Um conjunto de cadeias
Membro	Um objeto em um conjunto
Nó	Um ponto em um grafo
Operação booleana	Uma operação sobre valores booleanos
Par	Uma lista de dois elementos, também chamada 2-upla
Predicado	Uma função cujo contradomínio é {VERDADEIRO, FALSO}
Produto cartesiano	Uma operação sobre conjuntos formando um conjunto de todas as uplas de elementos dos respectivos conjuntos

0.3. DEFINIÇÕES, TEOREMAS, E PROVAS 13

Propriedade	Um predicado
Relação	Um predicado, mais tipicamente quando o domínio é um conjunto de k -uplas
Relação binária	Uma relação cujo domínio é um conjunto de pares
Relação de equivalência	Uma relação binária que é reflexiva, simétrica, e transitiva
Seqüência	Uma lista de objetos
Símbolo	Um membro de um alfabeto
União	Uma operação sobre conjuntos combinando todos os elementos em um único conjunto
Valor booleano	Os valores VERDADEIRO ou FALSO, frequentemente representados por 0 ou 1
Vértice	Um ponto em um grafo

0.3 Definições, teoremas, e provas

Teoremas e provas são o coração e a alma de matemática e definições são seu espírito. Essas três entidades são centrais para todo assunto matemático, incluindo o nosso.

Definições descrevem os objetos e noções que usamos. Uma definição pode ser simples, como na definição de *conjunto* dada neste capítulo, ou complexa como na definição de *segurança* em um sistema criptográfico. Precisão é essencial para qualquer definição matemática. Quando definindo algum objeto temos que deixar claro o que constitui aquele objeto e o que não.

Após termos definido vários objetos e noções, usualmente fazemos **enunciados matemáticos** sobre eles. Tipicamente um enunciado expressa que algum objeto tem uma certa propriedade. O enunciado pode ou não ser verdadeiro, mas como uma definição, ele tem que ser preciso. Não pode haver qualquer ambigüidade sobre seu significado.

Uma **prova** é um argumento lógico convincente de que um enunciado é verdadeiro. Em matemática um argumento tem que ser justo, ou seja, convincente em um sentido absoluto. Isso é bastante diferente da noção de prova que usamos na vida cotidiana ou na lei. Um julgamento por assassinato requer prova “além de qualquer dúvida razoável.” O peso da evidência pode compelir o júri a aceita a inocência ou a culpa do suspeito. Entretanto, evidência não desempenha nenhum papel em uma prova matemática. Um matemático demanda prova além de *qualquer* dúvida.

Um **teorema** é um enunciado matemático demonstrado verdadeiro. Geralmente reservamos o uso dessa palavra para enunciados de interesse especial. Ocionalmente provamos enunciados que são interessantes somente porque eles ajudam na prova de um outro enunciado mais significativo. Tais enunciados são chamados **lemas**. Ocionalmente um teorema ou sua prova podem nos permitir concluir facilmente aquele outro, ou outros, enunciado(s) relacionado(s). Esses enunciados são chamados **corolários** do teorema.

Encontrando provas

A única maneira de determinar a veracidade ou a falsidade de um enunciado matemático é uma prova matemática. Infelizmente, encontrar provas não é sempre fácil. Não pode ser reduzido a um simples conjunto de regras ou processos. Durante este curso, você será requisitado a apresentar provas de vários enunciados. Não desespere

só de pensar nisso! Muito embora ninguém tenha uma receita para produzir provas, algumas estratégias gerais úteis estão disponíveis.

Primeiro, leia cuidadosamente o enunciado que você deseja provar. Você entende toda a notação? Reescreva o enunciado com suas próprias palavras. Quebre-o e considere cada parte separadamente.

Às vezes as partes de um enunciado composto de muitas partes não são imediatamente evidentes. Um tipo de enunciado composto de muitas partes que ocorre frequentemente tem a forma “ P se e somente se Q ”, normalmente escrito “ P sse Q ”, onde ambos P e Q são enunciados matemáticos. Essa notação é uma abreviação para um enunciado de duas partes. A primeira parte é “ P somente se Q ” que significa: Se P é verdadeiro, então Q é verdadeiro, escrito $P \Rightarrow Q$. A segunda é “ P se Q ” que significa: Se Q é verdadeiro, então P é verdadeiro, escrito $P \Leftrightarrow Q$. A primeira dessas partes é a *direção de ida* do enunciado original e a segunda é a *direção de volta*. Escrevemos “ P se e somente se Q ” como $P \Leftrightarrow Q$. Para provar um enunciado dessa forma você tem que provar cada uma das duas direções. Frequentemente, uma dessas direções é mais fácil de provar que a outra.

Um outro tipo de enunciado composto de muitas partes enuncia que dois conjuntos A e B são iguais. A primeira parte enuncia que A é um subconjunto de B , e a segunda parte enuncia que B é um subconjunto de A . Portanto uma maneira comum de provar que $A = B$ é provar que todo membro de A é um membro de B e que todo membro de B é também um membro de A .

A seguir, quando você deseja provar um enunciado ou parte dele, tente obter um sentimento “lá de dentro” do por que ele deveria ser verdadeiro. Experimentar com exemplos é especialmente útil. Por conseguinte, se o enunciado diz que todos os objetos de um certo tipo têm uma propriedade específica, escolha uns poucos objetos daquele tipo e observe que eles de fato têm aquela propriedade. Após fazer isso, tente encontrar um objeto que falha em ter a propriedade, chamada um *contra-exemplo*. Se o enunciado de fato é verdadeiro, você não será capaz de encontrar um contra-exemplo. Ver onde você esbarra em dificuldade quando você tenta encontrar um contra-exemplo pode ajudar você a entender por que o enunciado é verdadeiro.

Exemplo 0.9

Suponha que você deseja provar o enunciado *para todo grafo G , a soma dos graus de todos os nós em G é um número par*.

Primeiro, pegue uns poucos grafos e observe esse enunciado em ação. Aqui estão alguns exemplos.

A seguir, tente encontrar um contra-exemplo, ou seja, um grafo no qual a soma é um número ímpar.

Você pode agora começar a ver por que o enunciado é verdadeiro e como prová-lo?

Se você ainda está emperrado tentando provar um enunciado, tente algo mais fácil. Tente provar um caso especial do enunciado. Por exemplo, se você está tentando provar que alguma propriedade é verdadeira para todo $k > 0$, primeiro tente prová-lo para $k = 1$. Se você conseguir, tente-o para $k = 2$, e assim por diante até que você possa

entender o caso mais geral. se um caso especial é difícil de provar, tente um caso especial diferente ou talvez um caso especial do caso especial.

Finalmente, quando você acredita que você encontrou a prova, você tem que escrevê-la apropriadamente. Uma prova bem escrita é uma seqüência de enunciados, no qual cada um segue por raciocínio simples dos enunciados anteriores na seqüência. Escrever cuidadosamente uma prova é importante, tanto para permitir que um leitor a entenda quanto para você ficar certo de que ela está livre de erros.

Aqui estão umas poucas dicas para produzir uma prova.

- *Seja paciente.* Encontrar provas leva tempo. Se você não vê como fazê-lo imediatamente, não se preocupe. Pesquisadores às vezes trabalham por semanas ou até anos para encontrar uma única prova.
- *Volte a ela.* Olhe para o enunciado que você deseja provar, pense nele um pouco, deixe-o, e então retorne alguns minutos ou horas depois. Dê uma chance para a parte inconsciente, intuitiva de sua mente trabalhar.
- *Seja claro e limpo.* Quando você está construindo sua intuição para o enunciado que você está tentando provar, use figuras ou textos simples, claras(os). Você está tentando desenvolver sua percepção sobre o enunciado, e desorganização atrapalha a percepção. Além do mais, quando você está escrevendo uma solução para uma outra pessoa ler, clareza e organização vai ajudar aquela pessoa a entendê-lo.
- *Seja conciso.* Brevidade ajuda a você expressar idéias de alto nível sem se perder em detalhes. Boa notação matemática é útil para expressar idéias concisamente. Mas tenha certeza de que incluiu o suficiente de seu raciocínio ao escrever uma prova de modo que o leitor possa facilmente entender o que você está tentando dizer.

Para praticar, vamos provar uma das leis de De Morgan.

Teorema 0.10

Para quaisquer dois conjuntos A e B , $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

Primeiro, o significado desse teorema está claro? Se você não entende o significado dos símbolos \cup ou \cap ou a barra superior, revise a discussão na página 4.

Para provar esse teorema temos que mostrar que os dois conjuntos $\overline{A \cup B}$ e $\overline{A} \cap \overline{B}$ são iguais. Lembre-se que podemos provar que dois conjuntos são iguais mostrando que todo membro de um conjunto também é membro do outro e vice versa. Antes de olhar para a prova abaixo, considere uns poucos exemplos e aí então tente prová-lo você mesmo.

Prova. Esse teorema enuncia que dois conjuntos, $\overline{A \cup B}$ e $\overline{A} \cap \overline{B}$, são iguais. Provamos essa asserção mostrando que todo elemento de um também é um elemento do outro, e vice versa.

Suponha que x seja um elemento de $\overline{A \cup B}$. Então x não está em $A \cup B$ da definição do complemento de um conjunto. Por conseguinte, x não está em A e x não está em B , da definição da união de dois conjuntos. Em outras palavras, x está em \overline{A} e x está em \overline{B} . Daí a definição da interseção de dois conjuntos mostra que x está em $\overline{A} \cap \overline{B}$.

Para a outra direção, suponha que x esteja em $\overline{A} \cap \overline{B}$. Então x está em ambos \overline{A} e \overline{B} . Por conseguinte x não está em A e x não está em B , e portanto não está na

união desses dois conjuntos. Daí x está no complemento da união desses conjuntos; em outras palavras, x está em $\overline{A \cup B}$ o que completa a prova do teorema.

Vamos tentar um outro.

Teorema 0.11

Em qualquer grafo G , a soma dos graus dos nós de G é um número par.

Prova. Toda aresta em G está conectada a dois nós. Cada aresta contribui com 1 para cada nó ao qual ela está conectada. Por conseguinte cada aresta contribui com 2 para a soma dos graus de todos os nós. Logo, se G contém e arestas, então a soma dos graus de todos os nós de G é $2e$, que é um número par.

0.4 Tipos de prova

Vários tipos de argumentos surgem frequentemente em provas matemáticas. Aqui, descrevemos uns poucos que normalmente ocorrem na teoria da computação. Note que uma prova pode conter mais que um tipo de argumento porque a prova pode conter nela várias subprovas diferentes.

Provas por construção

Muitos teoremas enunciam que um tipo específico de objeto existe. Uma maneira de provar um tal teorema é demonstrando como construir o objeto. Essa técnica é uma *prova por construção*.

Vamos usar uma prova por construção para provar o teorema a seguir. Definimos um grafo como sendo *k-regular* se todo nó no grafo tem grau k .

Teorema 0.12

Para cada número par n maior que 2, existe um grafo 3-regular com n nós.

Prova. Seja n um número par maior que 2. Construa um grafo $G = (V, E)$ com n nós da seguinte maneira. O conjunto de nós de G é $V = \{0, 1, \dots, n-1\}$, e o conjunto de arestas de G é o conjunto

$$\begin{aligned} E = & \{ \{i, i+1\} \mid \text{para } 0 \leq i \leq n-2 \} \cup \{ \{n-1, 0\} \} \\ & \cup \{ \{i, i+n/2\} \mid \text{para } 0 \leq i \leq n/2-1 \}. \end{aligned}$$

Desenhe os nós desse grafo escritos consecutivamente em torno da circunferência de um círculo. Nesse caso as arestas descritas na linha superior de E vão entre pares adjacentes ao redor do círculo. As arestas descritas na linha inferior de E vão entre nós em lados opostos no círculo. Essa figura mental claramente mostra que todo nó em G tem grau 3.

Prova por contradição

Em uma forma comum de argumento para provar um teorema, assumimos que o teorema é falso e aí então mostramos que esse suposição leva a uma consequência obviamente falsa, chamada uma contradição. Usamos esse tipo de raciocínio frequentemente na vida cotidiana, como no seguinte exemplo:

Exemplo 0.13

João vê Júlia, que acaba de entrar da rua. Observando que ela está completamente seca, ele sabe que não está chovendo. Sua “prova” de que não está chovendo é que, *se estivesse chovendo* (a suposição de que o enunciado é falso), *Júlia estaria molhada* (a consequência obviamente falsa). Por conseguinte não pode estar chovendo.

A seguir, vamos provar por contradição que a raiz quadrada de 2 é um número irracional. Um número é **racional** se ele é uma fração m/n onde m e n são inteiros; em outras palavras, um número racional é a *razão* dos inteiros m e n . Por exemplo, $2/3$ obviamente é um número racional. Um número é **irracional** se ele não é racional.

Teorema 0.14

$\sqrt{2}$ é irracional.

Prova. Primeiro, assumimos para mais adiante obter uma contradição que $\sqrt{2}$ é racional. Portanto

$$\sqrt{2} = \frac{m}{n},$$

onde tanto m quanto n são inteiros. Se ambos m e n são divisíveis pelo mesmo inteiro maior que 1, divida ambos por esse inteiro. Fazer isso não muda o valor da fração. Agora, nem m nem n podem ser números pares.

Multiplicamos ambos os lados da equação por n e obtemos:

$$n\sqrt{2} = m.$$

Elevamos ao quadrado ambos os lados e obtemos

$$2n^2 = m^2.$$

Devido ao fato de que m^2 é 2 vezes o inteiro n^2 , sabemos que m^2 é par. Por conseguinte, m também é par, pois o quadrado de um número ímpar sempre é ímpar. Portanto podemos escrever $m = 2k$ para algum inteiro k . Então, substituindo m por $2k$, obtemos

$$\begin{aligned} 2n^2 &= (2k)^2 \\ &= 4k^2. \end{aligned}$$

Dividindo ambos os lados por 2 obtemos

$$n^2 = 2k^2.$$

Mas esse resultado mostra que n^2 é par e portanto que n é par. Portanto estabelecemos que ambos m e n são pares. Mas tínhamos reduzido m e n de tal modo que eles *não* eram ambos pares, uma contradição.

.....

Prova por indução

Prova por indução é um método avançado de mostrar que todos os elementos de um conjunto infinito têm uma propriedade especificada. Por exemplo, podemos usar uma prova por indução para mostrar que uma expressão aritmética computa uma quantidade desejada para toda atribuição a suas variáveis ou que um programa funciona corretamente em todos os passos ou para todas as entradas.

Para ilustrar como a prova por indução funciona, vamos tomar o conjunto infinito como sendo os números naturais, $\mathcal{N} = \{1, 2, 3, \dots\}$, e digamos que a propriedade é chamada \mathcal{P} . Nossa objetivo é provar que $\mathcal{P}(k)$ é verdadeiro para cada número natural k . Em outras palavras, desejamos provar que $\mathcal{P}(1)$, assim como $\mathcal{P}(2)$, $\mathcal{P}(3)$, $\mathcal{P}(4)$, e assim por diante.

Toda prova por indução consiste de duas partes, o **passo da indução** e a **base**. Cada parte é uma prova individual em si própria. O passo da indução prova que para cada $i \geq 1$, se $\mathcal{P}(i)$ é verdadeiro, então $\mathcal{P}(i + 1)$ também o é. A base prova que $\mathcal{P}(1)$ é verdadeiro.

Quando tivermos provado ambas essas partes, o resultado segue, a saber, que $\mathcal{P}(i)$ é verdadeiro para todo i . Por que? Primeiro, sabemos que $\mathcal{P}(1)$ é verdadeiro porque a base sozinha prova isso. Segundo, sabemos que $\mathcal{P}(2)$ é verdadeiro porque o passo da indução prova que, se $\mathcal{P}(1)$ é verdadeiro então $\mathcal{P}(2)$ é verdadeiro, e já sabemos que $\mathcal{P}(1)$ é verdadeiro. Terceiro, sabemos que $\mathcal{P}(3)$ é verdadeiro porque o passo da indução prova que, se $\mathcal{P}(2)$ é verdadeiro então $\mathcal{P}(3)$ é verdadeiro, e já sabemos que $\mathcal{P}(2)$ é verdadeiro. Esse processo continua para todos os números naturais, mostrando que $\mathcal{P}(4)$ é verdadeiro, $\mathcal{P}(5)$ é verdadeiro, e assim por diante.

Uma vez que você entende o parágrafo precedente, você pode facilmente entender variações e generalizações da mesma idéia. Por exemplo, a base não necessariamente precisa começar com 1; ela pode começar com qualquer valor b . Nesse caso a prova por indução mostra que $\mathcal{P}(k)$ é verdadeiro para todo k que é pelo menos b .

No passo da indução a suposição de que $\mathcal{P}(i)$ é verdadeiro é chamada a **hipótese da indução**. Às vezes se ter uma hipótese da indução mais forte de que $\mathcal{P}(j)$ é verdadeiro para todo $j \leq i$ é útil. A prova por indução ainda funciona porque, quando desejamos provar que $\mathcal{P}(i + 1)$ é verdadeiro já provamos que $\mathcal{P}(j)$ é verdadeiro para todo $j \leq i$.

O formato para escrever uma prova por indução é como segue.

Base: Prove que $\mathcal{P}(1)$ é verdadeiro.

Passo da indução: Para cada $i \geq 1$, assuma que $\mathcal{P}(i)$ é verdadeiro e use essa hipótese para mostrar que $\mathcal{P}(i + 1)$ é verdadeiro.

Agora, vamos provar por indução a corretude da fórmula usada para calcular o tamanho das prestações da casa própria. Ao comprar uma casa, muitas pessoas tomam algum dinheiro emprestado necessário para a aquisição e pagam esse empréstimo durante um certo número de anos. Tipicamente, os termos de tal pagamento estipulam que uma quantidade fixa de dinheiro é paga cada mês para cobrir os juros, assim como parte da soma original, de modo que o total é pago em 30 anos. A fórmula para calcular o tamanho das prestações mensais é envolvida em mistério, mas na verdade é bastante simples. Isso afeta a vida de muitas pessoas, portanto você deve achá-lo interessante. Usamos indução para provar que ele funciona, tornando-o uma boa ilustração dessa técnica.

Primeiro, fixamos os nomes e significados de diversas variáveis. Seja P o *principal*, o valor do empréstimo original. Seja J a *taxa de juros* anual do empréstimo, onde $J = 0,06$ indica uma taxa de juros de 6%. Seja Y a prestação mensal. Por conveniência definimos uma outra variável M de J , para o multiplicador mensal. Trata-se da taxa na base da qual o empréstimo muda a cada mês devido à prestação mensal. Seja P_t o valor do empréstimo após o t -ésimo mês. Então $P_0 = P$ é o valor do empréstimo original, $P_1 = MP_0 - Y$ é o valor do empréstimo após um mês, $P_2 = MP_1 - Y$ é o empréstimo após dois meses, e assim por diante. Agora estamos prontos para enunciar e provar um teorema por indução sobre t que dá a fórmula para o valor de P_t .

Teorema 0.15

Para cada $t \geq 0$,

$$P_t = PM^t - Y \left(\frac{M^t - 1}{M - 1} \right).$$

Prova.

Base: Prove que a fórmula é verdadeira para $t = 0$. Se $t = 0$, então a fórmula enuncia que

$$P_0 = PM^0 - Y \left(\frac{M^0 - 1}{M - 1} \right).$$

Podemos simplificar o lado direito observando que $M^0 = 1$. Por conseguinte obtemos

$$P_0 = P,$$

que se verifica porque definimos P_0 como sendo P . Portanto provamos que a base da indução é verdadeira.

Passo da indução: Para cada $k \geq 0$ assuma que a fórmula é verdadeira para $t = k$ e mostre que ela é verdadeira para $t = k + 1$. A hipótese da indução enuncia que

$$P_k = PM^k - Y \left(\frac{M^k - 1}{M - 1} \right).$$

Nosso objetivo é provar que

$$P_{k+1} = PM^{k+1} - Y \left(\frac{M^{k+1} - 1}{M - 1} \right).$$

Fazemos isso com os seguintes passos. Primeiro, da definição de P_{k+1} a partir de P_k , sabemos que

$$P_{k+1} = P_k M - Y.$$

Por conseguinte, usando a hipótese da indução para calcular P_k ,

$$P_{k+1} = \left[PM^k - Y \left(\frac{M^k - 1}{M - 1} \right) \right] M - Y.$$

Multiplicando o M para dentro da expressão maior e reescrevendo Y dá

$$\begin{aligned} P_{k+1} &= PM^{k+1} - Y \left(\frac{M^{k+1} - M}{M - 1} \right) - Y \left(\frac{M - 1}{M - 1} \right) \\ &= PM^{k+1} - Y \left(\frac{M^{k+1} - 1}{M - 1} \right). \end{aligned}$$

Portanto a fórmula é correta para $t = k + 1$, o que prova o teorema.

.....

O Problema 0.13 pede que você use essa fórmula para calcular as verdadeiras prestações mensais.

.....

Exercícios

0.1 Examine as descrições formais de conjuntos abaixo de modo que você entenda quais membros eles contêm. Escreva uma descrição informal breve em português de cada conjunto.

- $\{1, 3, 5, 7, \dots\}$.
- $\{\dots, -4, -2, 0, 2, 4, \dots\}$.
- $\{n \mid n = 2m \text{ para algum } m \text{ em } \mathbb{N}\}$.
- $\{n \mid n = 2m \text{ para algum } m \text{ em } \mathbb{N}, \text{ e } n = 3k \text{ para algum } k \text{ em } \mathbb{N}\}$.
- $\{w \mid w \text{ é uma cadeia de } 0's \text{ e } 1's \text{ e } w \text{ é igual ao reverso de } w\}$.
- $\{n \mid n \text{ é um inteiro e } n = n + 1\}$.

0.2 Escreva descrições formais dos seguinte conjuntos:

- O conjunto contendo os números 1, 10, e 100.
- O conjunto contendo todos os inteiros que são maiores que 5.
- O conjunto contendo todos os números naturais que são menores que 5.
- O conjunto contendo a cadeia aba.
- O conjunto contendo a cadeia vazia.
- O conjunto contendo absolutamente nada.

0.3 Seja A o conjunto $\{x, y, z\}$ e B o conjunto $\{x, y\}$.

- A é um subconjunto de B ?
- B é um subconjunto de A ?
- O que é $A \cup B$?
- O que é $A \cap B$?
- O que é $A \times B$?
- Qual é o conjunto das partes de B ?

0.4 Se A tem a elementos e B tem b elementos, quantos elementos estão em $A \times B$? Explique sua resposta.

0.5 Se C é um conjunto com c elementos, quantos elementos estão em no conjunto das partes de C ? Explique sua resposta.

0.6 Seja X o conjunto $\{1, 2, 3, 4, 5\}$ e Y o conjunto $\{6, 7, 8, 9, 10\}$. A função unária $f : X \rightarrow Y$ e a função binária $g : X \times Y \rightarrow Y$ são descritas nas seguintes tabelas.

n	$f(n)$	g	6	7	8	9	10
1	6	1	10	10	10	10	10
2	7	2	7	8	9	10	6
3	6	3	7	7	8	8	9
4	7	4	9	8	7	6	10
5	6	5	6	6	6	6	6

- a. Qual é o valor de $f(2)$?
- b. Quais são o domínio e o contradomínio de f ?
- c. Qual é o valor de $g(2, 10)$?
- d. Quais são o domínio e o contradomínio de g ?
- e. Qual é o valor de $g(4, f(4))$?

0.7 Para cada item abaixo, dê uma relação que satisfaz a condição.

- a. Reflexiva e simétrica mas não transitiva.
- b. Reflexiva e transitiva mas não simétrica.
- c. Simétrica e transitiva mas não reflexiva.

0.8 Considere o grafo não-direcionado $G = (V, E)$ onde V , o conjunto de nós, é $\{1, 2, 3, 4\}$ e E , o conjunto de arestas, é $\{\{1, 2\}, \{2, 3\}, \{1, 3\}, \{2, 4\}, \{1, 4\}\}$. Desenhe o grafo G . Qual é o grau do nó 1? E do nó 3? Indique um caminho do nó 3 para o nó 4 sobre seu desenho do grafo de G .

0.9 Escreva uma descrição formal do grafo abaixo.

Problemas

0.10 Encontre o erro na seguinte prova de que $2 = 1$.

Considere a equação $a = b$. Multiplique ambos os lados por a para obter $a^2 = ab$. Subtraia b^2 de ambos os lados para obter $a^2 - b^2 = ab - b^2$. Agora fatore cada lado, $(a + b)(a - b) = b(a - b)$, e divida cada lado por $(a - b)$, para obter $a + b = b$. Finalmente, suponha que a e b sejam iguais a 1, o que mostra que $2 = 1$.

0.11 Encontre o erro na seguinte prova de todos os cavalos são da mesma cor.

AFIRMAÇÃO: Em qualquer conjunto de h cavalos, todos os cavalos são da mesma cor.

PROVA: Por indução sobre h .

Base: Para $h = 1$. Em qualquer conjunto contendo somente um cavalo, todos os cavalos claramente têm a mesma cor.

Passo da indução: Para $k \geq 1$ assuma que a afirmação seja verdadeira para $h = k$ e prove que ela é verdadeira para $h = k + 1$. Tome qualquer conjunto H de $k + 1$ cavalos. Mostramos que todos os cavalos nesse conjunto são da mesma cor. Remova um cavalo desse conjunto para obter o conjunto H_1 com apenas k cavalos. Pela hipótese da indução, todos os cavalos em H_1 são da mesma cor.

Agora recoloque o cavalo removido e remova um diferente para obter o conjunto H_2 . Pelo mesmo argumento, todos os cavalos em H_2 são da mesma cor. Por conseguinte todos os cavalos em H têm que ser da mesma cor, e a prova está completa.

- ***0.12** Teorema de Ramsey. Seja G um grafo. Um *clique* em G é um subgrafo no qual todo par de nós é conectado por uma aresta. Um *anti-clique*, também chamado *conjunto independente*, é um subgrafo no qual todo par de nós não está ligado por uma aresta. Mostre que todo grafo com n nós contém ou um clique ou um anti-clique com pelo menos $\frac{1}{2} \log_2 n$ nós.
- 0.13** Use o Teorema 0.15 para derivar uma fórmula para calcular o tamanho da prestação mensal da casa própria em termos do principal P , a taxa de juros J , e o número de pagamentos t . Assuma que, depois que t pagamentos tiverem sido feitos, o valor do empréstimo é reduzido a 0. Use a fórmula para calcular a quantidade em dólares de cada prestação mensal para um empréstimo de 30-anos com 360 prestações mensais sobre um empréstimo inicial no valor de 100.000 dólares com uma taxa de juros de 8%.

Parte Um:

Autômatos e Linguagens

Capítulo 1

Linguagens Regulares

A teoria da computação começa com uma pergunta: O que é um computador? É talvez uma pergunta boba, pois mesmo minha filha de 4 anos de idade sabe que essa coisa sobre a qual estou teclando é um computador. Mas esses computadores reais são bastante complicados—demasiado a ponto de nos permitir estabelecer uma teoria matemática manuseável sobre eles diretamente. Ao invés, usamos um computador idealizado chamado um *modelo computacional*. Como com qualquer modelo em ciência, um modelo computacional pode ser preciso de algumas formas mas talvez não em outras. Portanto usaremos vários modelos computacionais diferentes, dependendo das características sobre as quais desejamos focar. Começamos com o modelo mais simples, chamado *máquina de estados finitos* ou *autômato finito*.

1.1 Autômatos finitos

Autômatos finitos são bons modelos para computadores com uma quantidade de memória extremamente limitada. O que pode um computador fazer com uma memória tão pequena? Muitas coisas úteis! Na verdade, interagimos com tais computadores o tempo todo, pois eles residem no coração de vários dispositivos eletromecânicos.

Como mostrado nas figuras a seguir, o controlador para uma porta automática é um exemplo de tal dispositivo. Frequentemente encontradas em entradas e saídas de supermercados, portas automáticas abrem deslizando quando uma pessoa está se aproximando. Uma porta automática tem um tapete na frente para detectar a presença de uma pessoa que está próxima a atravessar a passagem. Um outro tapete está localizado atrás da passagem de modo que o controlador pode manter a porta aberta um tempo suficiente para que a pessoa atravesse toda a passagem e também de modo que a porta não atinja alguém que está atrás no momento que ela abre.

O controlador está em um dos dois estados: “ABERTA” ou “FECHADA,” representando a condição correspondente da porta. Como mostrado nas figuras a seguir, existem quatro condições possíveis: “FRENTE” (significando que uma pessoa está pisando no tapete na frente da passagem), “ATRÁS” (significando que uma pessoa está

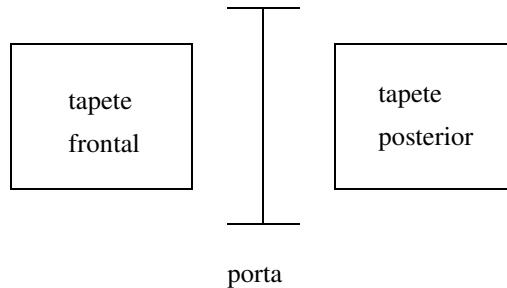


Figura 1.1: Vista superior de uma porta automática

pisando no tapete após a passagem), “AMBOS” (significando que as pessoas estão pisando em ambos os tapetes), e “NENHUM” (significando que ninguém está pisando em qualquer dos tapetes).



Figura 1.2: Diagrama de estados para o controlador de porta automática

estado		NENHUM	FRENTE	ATRÁS	AMBOS
	FECHADA	FECHADA	ABERTA	FECHADA	FECHADA
ABERTA	FECHADA	ABERTA	ABERTA	ABERTA	ABERTA

Figura 1.3: Tabela de transição de estado para o controlador de porta automática

O controlador move de estado para estado, dependendo da entrada que ele recebe. Quando no estado “FECHADA” e recebendo uma entrada NENHUM ou ATRÁS, ele permanece no estado FECHADA. Adicionalmente, se a entrada AMBOS é recebida, ele permanece FECHADA porque a porta corre o risco de atingir alguém sobre o tapete de trás. Mas se a entrada FRENTE chega, ele move para o estado ABERTA. No estado ABERTA, se a entrada FRENTE, ATRÁS, ou AMBOS é recebida, ele permanece em ABERTA. Se a entrada NENHUM chega, ele retorna a FECHADA.

Por exemplo, um controlador pode começar no estado FECHADA e receber a seguinte série de sinais de entrada: FRENTE, ATRÁS, NENHUM, FRENTE, AMBOS, NENHUM, ATRÁS, NENHUM. Ele então passaria pela série de estados: FE-

CHADA (começando), ABERTA, ABERTA, FECHADA, ABERTA, ABERTA, FECHADA, FECHADA, FECHADA.

Pensando num controlador de porta automática como um autômato finito, é útil porque sugere formas padronizadas de representação como nas Figuras 1.2 e 1.3. Esse controlador é um computador que tem somente um bit de memória, capaz de gravar em quais dos dois estados o controlador está. Outros dispositivos comuns têm controladores com memórias algo maiores. Em um controlador de elevador um estado pode representar o andar no qual o elevador está e as entradas pode ser os sinais recebidos dos botões. Esse computador pode precisar de vários bits para guardar essa informação. Controladores para vários dispositivos domésticos tais como lavadoras de prato e termostatos eletrônicos, assim como peças de relógios digitais e calculadoras, são exemplos adicionais de computadores com memórias limitadas. O desenho de tais dispositivos requer que se mantenha em mente a metodologia e a terminologia de autômatos finitos.

Autômatos finitos e suas contrapartidas probabilísticas *cadeias de Markov* são ferramentas úteis quando estamos tentando reconhecer padrões em dados. Esses dispositivos são usados em processamento de voz e em reconhecimento de caracteres óticos. Cadeias de Markov têm sido usadas para modelar e prever mudanças de preço em mercados financeiros.

Agora vamos dar uma olhada mais próxima em autômatos finitos sob uma perspectiva matemática. Desenvolveremos uma definição precisa de um autômato finito, terminologia para descrever e manipular autômatos finitos, e resultados teóricos que descrevem seu poder e suas limitações. Além de nos dar um entendimento mais claro do que são autômatos finitos, e do que eles podem e não podem fazer, o desenvolvimento teórico nos permite praticar e nos tornar mais confortáveis com definições matemáticas, teoremas, e provas em um cenário relativamente simples.

Ao começar a descrever a teoria matemática de autômatos finitos, fazemos isso no nível abstrato, sem referência a qualquer aplicação específica. A seguinte figura mostra um autômato finito chamado M_1 .

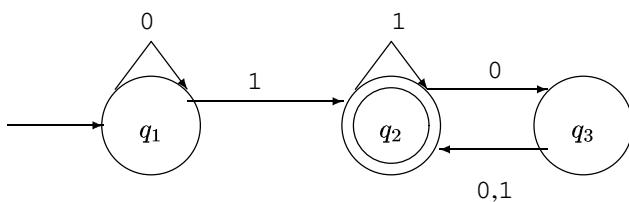


Figura 1.4: Um autômato finito chamado M_1 que tem três estados

A Figura 1.4 é chamada *diagrama de estado* de M_1 . O autômato tem três *estados*, rotulados q_1, q_2 e q_3 . O *estado inicial*, q_1 , é indicado pela seta apontando para ele a partir do nada. O *estado de aceitação*, q_2 , é aquele com um duplo círculo. As setas saindo de um estado para outro são chamadas *transições*.

Quando esse autômato recebe uma cadeia de entrada tal como 1101, ele processa essa cadeia e produz uma saída. A saída é *aceita* ou *rejeita*. Consideraremos apenas esse tipo de saída sim/não por enquanto de modo a manter as coisas simples. O processamento começa no estado inicial de M_1 . O autômato recebe os símbolos da cadeia

de entrada um por um da esquerda para a direita. Após ler cada símbolo, M_1 move de um estado para outro por meio da transição que tem aquele símbolo como seu rótulo. Quando ele lê o último símbolo, M_1 produz sua saída. A saída é *aceita* se M_1 está agora em um estado de aceitação e *rejeita* se não está.

Por exemplo, quando alimentamos a cadeia de entrada 1101 na máquina M_1 da Figura 1.4, o processamento procede da seguinte forma:

1. começa no estado q_1 ;
2. lê 1, segue transição de q_1 para q_2 ;
3. lê 1, segue transição de q_2 para q_2 ;
4. lê 0, segue transição de q_2 para q_3 ;
5. lê 1, segue transição de q_3 para q_2 ;
6. *aceita* porque M_1 está em um estado de aceitação q_2 no final da entrada.

Experimentando com essa máquina sobre uma variedade de cadeias de entrada revela que ela aceita as cadeias 1, 01, 11, e 01010101. Na verdade, M_1 aceita qualquer cadeia que termina com um 1, pois ela vai para seu estado de aceitação q_2 sempre que ela lê o símbolo 1. Adicionalmente, ela aceita as cadeias 100, 0100, 110000, e 010100000, e qualquer cadeia que termine com um número par de 0's após o último 1. Ela rejeita outras cadeias, tais como 0, 10, 101000. Você pode descrever a linguagem consistindo de todas as cadeias que M_1 aceita? Faremos isso brevemente.

Definição formal de um autômato finito

Na seção precedente usamos diagramas de estado para introduzir autômatos finitos. Agora definimos autômatos finitos formalmente. Embora diagramas de estado sejam mais fáceis de compreender intuitivamente, precisamos de uma definição formal também, por duas razões específicas.

Primeiro, uma definição formal é precisa. Ela resolve quaisquer incertezas sobre o que é permitido num autômato finito. Se você estivesse incerto sobre se autômatos finitos pudessem ter 0 estados de aceitação ou se eles têm que ter exatamente uma transição saindo de todo estado para cada símbolo de entrada possível, você poderia consultar a definição formal e verificar que a resposta é sim em ambos os casos. Segundo, uma definição formal provê notação. Boa notação ajuda a você pensar e expressar seus pensamentos claramente.

A linguagem de uma definição formal é um tanto misteriosa, tendo alguma semelhança com a linguagem de um documento legal. Ambos necessitam ser precisos, e todo detalhe deve ser explicitado.

Um autômato finito tem várias partes. Tem um conjunto de estados e regras para ir de um estado para outro, dependendo do símbolo de entrada. Tem um alfabeto de entrada que indica os símbolos de entrada permitidos. Tem um estado inicial e um conjunto de estados de aceitação. A definição formal diz que um autômato finito é uma lista daqueles cinco objetos: conjunto de estados, alfabeto de entrada, regras para movimentação, estado inicial, e estados de aceitação. Em linguagem matemática uma lista de cinco elementos é frequentemente chamada 5-upla.

Usamos algo chamado de uma *função de transição*, frequentemente denotado por δ , para definir as regras para movimentação. Se o autômato finito tem uma seta de um

estado x para um estado y rotulada com o símbolo de entrada 1, isso significa que, se o autômato está no estado x quando ele lê um 1, ele então move para o estado y . Podemos indicar a mesma coisa com a função de transição dizendo que $\delta(x, 1) = y$. Essa notação é uma espécie de abreviação matemática. Juntando tudo chegamos na definição formal de autômatos finitos.

Definição 1.1

Um **autômato finito** é uma 5-upla $(Q, \Sigma, \delta, q_0, F)$ onde

1. Q é um conjunto finito chamado de **os estados**,
2. Σ é um conjunto finito chamado de **o alfabeto**,
3. $\delta : Q \times \Sigma \rightarrow Q$ é a **função de transição**,¹
4. $q_0 \in Q$ é o **estado inicial**, e
5. $F \subseteq Q$ é o **conjunto de estados de aceitação**.²

A definição formal descreve precisamente o que queremos dizer por um autômato finito. Por exemplo, voltando à questão anterior sobre se 0 estados de aceitação é permitível, você pode ver que fazendo F ser o conjunto vazio \emptyset resulta em 0 estados de aceitação, o que é permitível. Além do mais a função de transição δ especifica exatamente um estado para cada combinação possível de um estado e um símbolo de entrada. Isso responde à nossa outra questão afirmativamente, mostrando que exatamente uma seta de transição sai de todo estado para cada símbolo de entrada possível.

Podemos usar a notação da definição formal para descrever autômatos finitos individualmente especificando cada uma das partes listadas na Definição 1.1. Por exemplo, vamos retornar ao autômato finito M_1 na Figura 1.4.

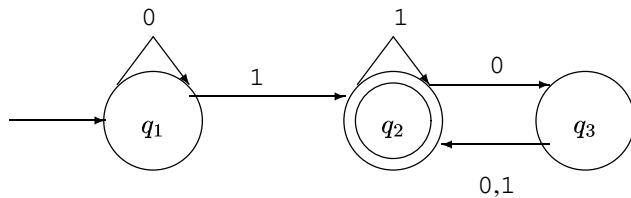


Figura 1.5: O autômato finito M_1

Podemos descrever M_1 formalmente escrevendo $M_1 = (Q, \Sigma, \delta, q_1, F)$, onde

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0, 1\}$,
3. δ é descrita como

¹Remeta-se de volta à página 7 se você está incerto sobre o significado de $\delta : Q \times \Sigma \rightarrow Q$.

²Estados de aceitação às vezes são chamados de **estados finais**.

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 é o estado inicial, e

5. $F = \{q_2\}$.

Se A é o conjunto de todas as cadeias que a máquina M aceita, dizemos que A é a *linguagem da máquina* M e escrevemos $L(M) = A$. Dizemos que M **reconhece** A ou que M **aceita** A . Devido ao fato de que o termo *aceita* tem significados diferentes quando nos referimos a máquinas aceitando cadeias e máquinas aceitando linguagens, preferimos o termo *reconhece* para linguagens de modo a evitar confusão.

Uma máquina pode aceitar diversas cadeias, mas ela sempre reconhece apenas uma linguagem. Se a máquina não aceita nenhuma cadeia, ela ainda reconhece uma linguagem, a saber a linguagem vazia \emptyset .

Em nosso exemplo, faça

$$\begin{aligned} A = \{w \mid w \text{ contém pelo menos um } 1 \text{ e} \\ \text{um número par de } 0's \text{ segue o último } 1\} \end{aligned}$$

Então $L(M_1) = A$, ou equivalentemente, M_1 reconhece A .

Exemplos de autômatos finitos

O diagrama abaixo é o diagrama de estados do autômato finito M_2 .

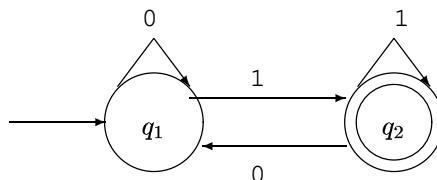


Figura 1.6: Diagrama de estados do autômato finito M_2 de dois estados

Na descrição formal $M_2 = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$. A função de transição δ é

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

Lembre-se que o diagrama de estados de M_2 e a descrição formal de M_2 contêm a mesma informação, apenas de forma diferente. Você pode sempre ir de uma para a outra se necessário.

Uma boa maneira de começar entendendo qualquer máquina é testá-la com algumas cadeias de entrada. Quando você faz esses “experimentos” para ver como a máquina está funcionando, seu método de funcionamento frequentemente se torna aparente. Sobre a cadeia de amostra 1101 a máquina M_2 começa no seu estado inicial q_1 e procede

primeiro para o estado q_2 após ler o primeiro 1, e então para os estados q_2 , q_1 , e q_2 após ler 1, 0, e 1. A cadeia é aceita porque o estado q_2 é um estado de aceitação. Mas a cadeia 110 deixa M_2 no estado q_1 , portanto ela é rejeitada. Após tentar alguns poucos exemplos mais, você verá que M_2 aceita todas as cadeias que terminam com um 1. Por conseguinte $L(M_2) = \{w \mid w \text{ termina com um } 1\}$.

Exemplo 1.3

Considere o autômato finito M_3 .

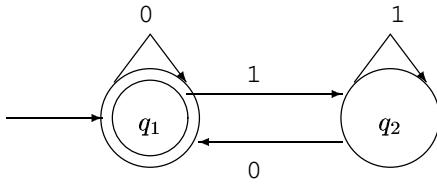


Figura 1.7: Diagrama de estados do autômato finito M_3 de dois estados

A máquina M_3 é semelhante a M_2 exceto pela localização do estado final. Como de costume, a máquina aceita todas as cadeias que a deixam em um estado de aceitação quando ela termina de ler. Note que, devido ao fato de que o estado inicial é também um estado de aceitação, M_3 aceita a cadeia vazia ε . Assim que a máquina começa a ler a cadeia vazia ela já está no fim, portanto se o estado inicial é um estado de aceitação, ε é aceita. Além da cadeia vazia, essa máquina aceita qualquer cadeia terminando com um 0. Aqui,

$$L(M_3) = \{w \mid w \text{ é a cadeia vazia } \varepsilon \text{ ou termina com um } 0\}.$$

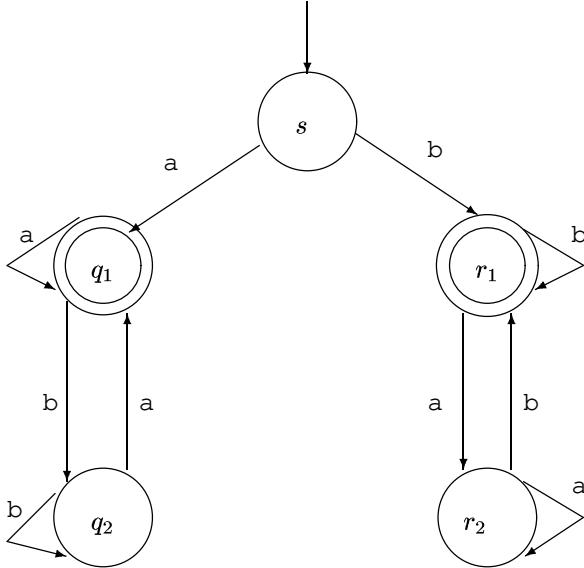
Exemplo 1.4

Considere o autômato finito M_4 .

M_4 tem dois estados de aceitação, q_1 e r_1 e opera sobre o alfabeto $\Sigma = \{a, b\}$. Um pouco de experimentação mostra que ela aceita as cadeias a , b , aa , bb , e bab , mas não as cadeias ab , ba , ou $bbba$. Essa máquina começa no estado s , e depois que ela lê o primeiro símbolo na entrada, ela vai para a esquerda para os estados q ou para a direita para os estados r . Em qualquer dos casos ela nunca pode retornar ao estado inicial (em contraste com o caso anterior), pois ela não tem maneira de sair de qualquer outro estado e voltar para s . Se o primeiro símbolo na cadeia de entrada é a , então ela vai para a esquerda e aceita quando a cadeia termina com um a . Similarmente, se o primeiro símbolo é um b , a máquina vai para a direita e aceita quando a cadeia termina em b . Portanto M_4 aceita todas as cadeias que começam e terminam com a , ou que começam e terminam com b . Em outras palavras, M_4 aceita cadeias que começam e terminam com o mesmo símbolo.

Exemplo 1.5

O diagrama de estados a seguir mostra a máquina M_5 , que tem um alfabeto de entrada de quatro símbolos $\Sigma = \{\langle ZERA \rangle, 0, 1, 2\}$. Tratamos $\langle ZERA \rangle$ como um único símbolo.

Figura 1.8: Autômato finito M_4

M_5 mantém um contador para a soma dos símbolos numéricos de entrada que ela lê, módulo 3. Toda vez que ela recebe o símbolo ⟨ZERA⟩ ela recoloca o contador em 0. Ela aceita se a soma é 0, módulo 3, ou em outras palavras, se a soma é um múltiplo de 3.

Exemplo 1.6

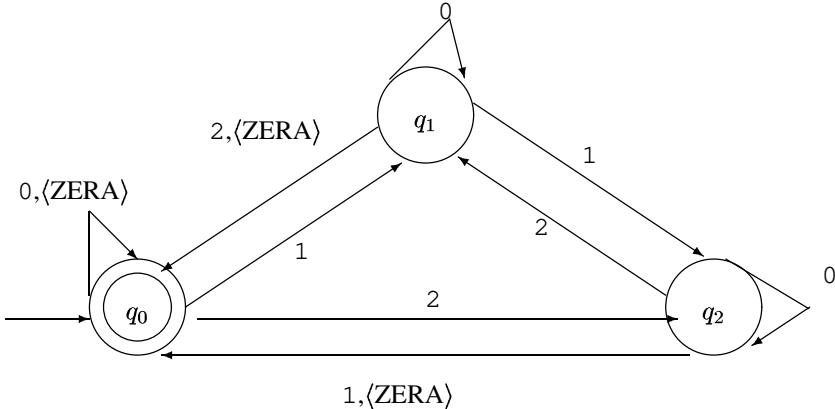
Descrever um autômato finito por diagrama de estados não é possível em alguns casos. Isso pode ocorrer quando o diagrama seria grande demais para desenhar ou se, como nesse exemplo, a descrição depende de algum parâmetro não-especificado. Nesses casos recorremos a uma descrição formal para especificar a máquina.

Considere uma generalização do Exemplo 1.5 usando os mesmos quatro símbolos do alfabeto Σ . Para cada $i \geq 1$ seja A_i a linguagem de todas as cadeias onde a soma dos números é um múltiplo de i , exceto que a soma é zerada sempre que o símbolo ⟨ZERA⟩ aparece. Para cada A_i damos um autômato finito B_i reconhecendo A_i . Descrevemos a máquina B_i formalmente da seguinte maneira: $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, onde Q_i é o conjunto de i estados $\{q_0, q_1, q_2, \dots, q_{i-1}\}$, e montamos a função de transição δ_i de modo que para cada j , se B_i está em q_j , a soma corrente é j , módulo i . Para cada q_j faça

$$\begin{aligned} \delta_i(q_j, 0) &= q_j \\ \delta_i(q_j, 1) &= q_k \text{ onde } k = j + 1 \text{ módulo } i, \\ \delta_i(q_j, 2) &= q_k \text{ onde } k = j + 2 \text{ módulo } i, \text{ e} \\ \delta_i(q_j, \langle \text{ZERA} \rangle) &= q_0. \end{aligned}$$

Definição formal de computação

Até agora descrevemos autômatos finitos informalmente, usando diagramas de estado, e com uma definição formal, como uma 5-upla. A descrição informal é mais fácil de compreender inicialmente, mas a definição formal é útil pois torna a noção totalmente


 Figura 1.9: Autômato finito M_5

precisa, resolvendo quaisquer ambigüidades que podem ter ocorrido na descrição informal. A seguir fazemos o mesmo para uma computação de um autômato finito. Já temos uma idéia informal da maneira com que ele computa, e agora a formalizamos matematicamente.

Seja $M = (Q, \Sigma, \delta, q_0, F)$ um autômato finito e $w = w_1 w_2 \cdots w_n$ uma cadeia sobre o alfabeto Σ . Então M **aceita** w se uma seqüência de estados r_0, r_1, \dots, r_n existe em Q com as seguintes três condições:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ para $i = 0, \dots, n - 1$, e
3. $r_n \in F$.

Condição 1 diz que a máquina começa no estado inicial. Condição 2 diz que a máquina vai de estado para estado conforme a função de transição. Condição 3 diz que a máquina aceita sua entrada se ela termina em um estado de aceitação. Dizemos que M **reconhece a linguagem** A se $A = \{w \mid M \text{ aceita } w\}$.

Definição 1.7

Uma linguagem é chamada de uma **linguagem regular** se algum autômato finito a reconhece.

Exemplo 1.8

Tome a máquina M_5 do Exemplo 1.5. Seja w a cadeia

$$10\langle\text{ZERA}\rangle22\langle\text{ZERA}\rangle012$$

Então M_5 aceita w conforme a definição formal de computação porque a seqüência de estados em que ela entra quando computando sobre w é

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

que satisfaz as três condições. A linguagem de M_5 é

$$L(M_5) = \{w \mid \begin{array}{l} \text{a soma dos símbolos em } w \text{ é 0 módulo 3,} \\ \text{exceto que } \langle\text{ZERA}\rangle \text{ zera o contador}\}.$$

Como M_5 reconhece essa linguagem, ela é uma linguagem regular.

Projetando autômatos finitos

Seja de um autômato ou de um trabalho artístico, projetar é um processo criativo. Como tal ele não pode ser reduzido a uma simples receita ou fórmula. Entretanto, você poderia achar uma determinada abordagem benéfica quando projetando vários tipos de autômatos. Isto é, ponha-se no lugar da máquina que você está tentando projetar e então veja como você iria realizar a tarefa da máquina. Fazendo de conta que você é a máquina é um truque psicológico que ajuda a engajar sua mente inteira no processo de projetar.

Vamos projetar um autômato finito usando o método do “leitor como autômato” que acaba de ser descrito. Suponha que lhe é dada uma linguagem e você deseja projetar um autômato finito que a reconheça. Fazendo de conta ser o autômato, você recebe uma cadeia de entrada e tem que determinar se ela é um membro da linguagem que o autômato deve reconhecer. Você vê os símbolos na cadeia um por um. Após cada símbolo você tem que decidir se a cadeia vista até então está na linguagem. A razão é que você, como a máquina, não sabe quando o final da cadeia está vindo, daí você tem que estar sempre pronto com a resposta.

Primeiro, de modo a tomar essas decisões, você tem que identificar o que você precisa memorizar sobre a cadeia à medida que você está lendo. Por que não simplesmente memorizar tudo o que você viu? Tenha em mente que você está fazendo de conta que é um autômato finito e que esse tipo de máquina tem apenas uma quantidade finita de estados, o que significa uma memória finita. Imagine que a entrada seja extremamente longa, digamos, daqui até a lua, de modo que você não poderia de forma alguma memorizar a coisa inteira. Você tem uma memória finita, digamos, uma única folha de papel, que tem uma capacidade de memória limitada. Felizmente, para muitas linguagens você não precisa memorizar a entrada inteira. Você somente necessita de memorizar uma certa informação crucial. Exatamente que informação é crucial depende da linguagem específica considerada.

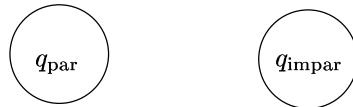
Por exemplo, considere que o alfabeto é $\{0, 1\}$ e que a linguagem consiste de todas as cadeias com um número ímpar de 1's. Você deseja construir um autômato finito E_1 para reconhecer essa linguagem. Fazendo de conta que é o autômato, você começa obtendo uma cadeia de entrada de 0's e 1's símbolo por símbolo. Você precisa memorizar a cadeia inteira vista até então de modo a determinar se o número de 1's é ímpar? Claro que não. Simplesmente memorize se o número de 1's vistos até então é par ou ímpar e mantenha um registro dessa informação à medida que você lê novos símbolos. Se você ler um 1, inverta a resposta, mas se você ler um 0, deixe a resposta do jeito que está.

Mas como isso ajuda a você projetar E_1 ? Uma vez que você determinou a informação necessária para memorizar a cadeia como ela está sendo lida, você representa essa informação como uma lista finita de possibilidades. Nessa instância, as possibilidades seriam

1. par até agora, e
2. ímpar até agora.

Então você associa um estado a cada uma das possibilidades. Esses são os estados de E_1 , como mostrado na figura seguinte.

A seguir, você associa as transições vendo como ir de uma possibilidade para outra ao ler um símbolo. Portanto, se o estado q_{par} representa a possibilidade par e o estado


 Figura 1.10: Os dois estados q_{par} e q_{impar}

q_{impar} representa a possibilidade ímpar, você faria com que as transições trocassem de estado sobre um 1 e permanecessem onde estavam sobre um 0, como mostrado na figura seguinte.

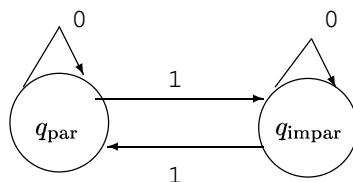


Figura 1.11: Transições dizendo como as possibilidades se rearranjam

A seguir, você marca como estado inicial o estado correspondente à possibilidade associada com ter visto 0 símbolos até então (a cadeia vazia ε). Neste caso o estado inicial corresponde ao estado q_{par} porque 0 é um número par. Por último, marque os estados de aceitação como sendo aqueles correspondentes às possibilidades nas quais você deseja aceitar a cadeia de entrada. Marque q_{impar} como sendo um estado de aceitação porque você deseja aceitar quando você tiver visto um número ímpar de 1's. Essas adições são mostradas na figura a seguir.

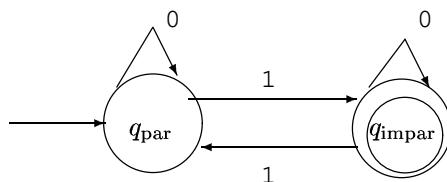


Figura 1.12: Adicionando os estados inicial e de aceitação

Exemplo 1.9

Esse exemplo mostra como desenhar um autômato finito E_2 para reconhecer a linguagem regular de todas as cadeias que contêm a cadeia 001 como uma subcadeia. Por exemplo, 0010, 1001, 001, e 111111001111 estão todas na linguagem, mas 11 e 0000 não estão. Como você reconheceria essa linguagem se você fosse fazer de conta que era E_2 ? À medida que símbolos chegam, você inicialmente pularia sobre todos os 1's. Se você chegasse a um 0, então você anotaria que você pode ter acabado de ver o primeiro dos três símbolos no padrão 001 que você está buscando. Se nesse ponto você vê um 1, houve poucos 0's, portanto você volta a pular sobre os 1's. Mas se você vê um 0 naquele ponto, você deve memorizar que você acabou de ver dois

símbolos do padrão. Agora você simplesmente precisa continuar a varrer até que você veja um 1. Se você o encontrar, memorize que você foi bem sucedido em encontrar o padrão e continue a ler a cadeia de entrada até que você chegue ao fim.

Portanto existem quatro possibilidades: Você

1. não chegou a ver quaisquer símbolos do padrão,
2. acabou de ver um 0,
3. acabou de ver 00, ou
4. viu o padrão inteiro 001.

Associe os estados q , q_0 , q_{00} , e q_{001} a essas possibilidades. Você pode associar as transições observando que de q lendo um 1 você permanece em q , mas lendo um 0 você move para q_0 . Em q_0 lendo um 1 você retorna a q , mas lendo um 0 você move para q_{00} . Em q_{00} lendo um 1 você move para q_{001} , mas lendo um 0 deixa você em q_{00} . Finalmente, em q_{001} lendo um 0 ou um 1 deixa você em q_{001} . O estado inicial é q , e o único estado de aceitação é q_{001} , como mostrado na figura seguinte.

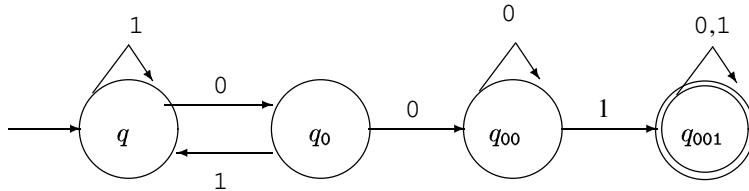


Figura 1.13: Aceita cadeias contendo 001

As operações regulares

Nas duas seções precedentes introduzimos e definimos autômatos finitos e linguagens regulares. Agora começamos a investigar suas propriedades. Fazer assim ajudará a desenvolver uma caixa de ferramentas de técnicas para usar quando você projeta autômatos para reconhecer linguagens específicas. A caixa de ferramentas também incluirá maneiras de provar que certas outras linguagens são não-regulares (i.e., além da capacidade de autômatos finitos).

Em aritmética, os objetos básicos são números e as ferramentas são operações para manipulá-los, tais como $+$ e \times . Na teoria da computação os objetos são linguagens e as ferramentas incluem operações especificamente desenhadas para manipulá-las. Definimos três operações sobre linguagens, chamadas de as **operações regulares**, e as usamos para estudar propriedades das linguagens regulares.

Definição 1.10

Sejam A e B linguagens. Definimos as operações regulares **união**, **concatenação**, e **estrela** da seguinte forma.

- **União:** $A \cup B = \{x \mid x \in A \text{ ou } x \in B\}$.
- **Concatenação:** $A \circ B = \{xy \mid x \in A \text{ e } y \in B\}$.

- **Estrela:** $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ e cada } x_i \in A\}$.

Você já está familiar com a operação de união. Ela simplesmente toma todas as cadeias em ambas A e B e junta todas em uma linguagem.

A operação de concatenação é um pouco mais complicada. Ela acrescenta uma cadeia de A na frente de uma cadeia de B de todas as maneiras possíveis para obter as cadeias na nova linguagem.

A operação estrela é um pouco diferente das outras duas porque ela se aplica a uma única linguagem ao invés de duas. Ou seja, a operação estrela é uma **operação unária** ao invés de uma **operação binária**. Ela funciona juntando um número qualquer de cadeias de A para obter uma cadeia na nova linguagem. Devido ao fato de que o termo “um número qualquer” inclui 0 como uma possibilidade, a cadeia vazia ε é sempre um membro de A^* , independente do que A seja.

Exemplo 1.11

Seja o alfabeto Σ constituído das 23 letras $\{a, b, \dots, z\}$. Se $A = \{\text{legal}, \text{ruim}\}$ e $B = \{\text{garoto}, \text{garota}\}$, então

$$\begin{aligned} A \cup B &= \{\text{legal}, \text{ruim}, \text{garoto}, \text{garota}\}, \\ B \circ A &= \{\text{garotolegal}, \text{garotalegal}, \text{garotoruim}, \text{garotaruim}\}, \text{ e} \\ A^* &= \{\varepsilon, \text{legal}, \text{ruim}, \text{legallegal}, \text{legalruim}, \text{ruimlegal}, \text{ruimruim}, \\ &\quad \text{legallegallegal}, \text{legallegalruim}, \text{legalruimlegal}, \text{legalruimruim}\}. \end{aligned}$$

Seja $\mathcal{N} = \{1, 2, 3, \dots\}$ o conjunto dos números naturais. Quando dizemos que \mathcal{N} é *fechado sob multiplicação* queremos dizer que, para quaisquer x e y em \mathcal{N} , o produto $x \times y$ também está em \mathcal{N} . Em geral, uma coleção de objetos é **fechada** sob alguma operação se aplicando-se aquela operação a membros da coleção retorna um objeto ainda na coleção. Mostramos que a coleção de linguagens regulares é fechada sob todas as três operações regulares. Na Seção 1.3 mostramos que essas são ferramentas úteis para manipulação de linguagens regulares e para o entendimento do poder dos autômatos finitos. Começamos com a operação de união.

Teorema 1.12

A classe das linguagens regulares é fechada sob a operação de união.

Em outras palavras, se A_1 e A_2 são linguagens regulares, então $A_1 \cup A_2$ também o é.

Idéia da prova. Temos duas linguagens regulares A_1 e A_2 e desejamos mostrar que $A_1 \cup A_2$ também é regular. Como A_1 e A_2 são regulares, sabemos que algum autômato finito M_1 reconhece A_1 e algum autômato finito M_2 reconhece A_2 . Para provar que $A_1 \cup A_2$ é regular exibimos um autômato finito, chame-o M , que reconhece $A_1 \cup A_2$.

Essa é uma prova por construção. Construímos M a partir de M_1 e M_2 . A máquina M tem que aceitar sua entrada exatamente quando uma das duas M_1 ou M_2 aceitar de modo a reconhecer a linguagem da união. Ela funciona *simulando* ambas M_1 e M_2 e aceitando se uma das duas simulações aceita.

Como podemos fazer a máquina M simular M_1 e M_2 ? Talvez ela primeiramente deva simular M_1 sobre a entrada e então simular M_2 sobre a entrada. Mas temos que ser cuidadosos aqui! Uma vez que os símbolos da entrada são lidos e usados para simular M_1 , não podemos “reenrolar a fita de entrada” para tentar a simulação sobre M_2 . Precisamos de uma outra abordagem.

Faça de conta que você é M . À medida que os símbolos de entrada chegam um por um, você simula ambas M_1 e M_2 simultaneamente. Dessa maneira somente uma

passagem sobre a entrada é necessária. Mas você pode manter o registro de ambas as simulações com memória finita? Tudo o que você precisa memorizar é o estado em que cada máquina estaria se ela tivesse lido até aquele ponto na entrada. Por conseguinte você precisa memorizar um par de estados. Quantos pares possíveis existem? Se M_1 tem k_1 estados e M_2 tem k_2 estados, o número de pares de estados, um de M_1 e o outro de M_2 , é o produto $k_1 \times k_2$. Esse produto será o número de estados em M , um para cada par. As transições de M vão de par para par, atualizando o estado corrente para ambas M_1 e M_2 . Os estados de aceitação de M são aqueles pares nos quais M_1 ou M_2 está num estado de aceitação.

Prova.

Suponha que M_1 reconhece A_1 , onde $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, e
 M_2 reconhece A_2 , onde $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$

Construa M para reconhecer $A_1 \cup A_2$, onde $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ e } r_2 \in Q_2\}$.

Esse conjunto é o *produto cartesiano* dos conjuntos Q_1 e Q_2 , e é escrito $Q_1 \times Q_2$. É o conjunto de todos os pares, o primeiro de Q_1 e o segundo de Q_2 .

2. Σ , o alfabeto, é o mesmo que em M_1 e M_2 . Neste teorema e em todos os teoremas subsequentes similares, assumimos por simplicidade que ambos M_1 e M_2 têm o mesmo alfabeto de entrada Σ . O teorema permanece verdadeiro se eles tiverem alfabetos diferentes, Σ_1 e Σ_2 . Modificariamós então a prova para fazer $\Sigma = \Sigma_1 \cup \Sigma_2$.

3. δ , a função de transição, é definida como segue. Para cada $(r_1, r_2) \in Q$, e cada $a \in \Sigma$, faça

$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$

Donde δ pega um estado de M (que na verdade é um par de estados de M_1 e M_2), juntamente com um símbolo de entrada, e retorna o estado seguinte de M .

4. q_0 é o par (q_1, q_2) .

5. F é o conjunto dos pares nos quais um dos membros é um estado de aceitação de M_1 ou M_2 .

Podemos escrevê-lo como

$$F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ ou } r_2 \in F_2\}.$$

Essa expressão é a mesma que $F = (F_1 \times Q_1) \cup (Q_2 \times F_2)$. (Note que ela *não* é a mesma que $F = F_1 \times F_2$. O que essa última nos daria diferentemente da primeira?³)

Isso conclui a construção do autômato finito M que reconhece a união de A_1 e A_2 . Essa construção é bastante simples, e por conseguinte sua corretude é evidente da estratégia que é descrita na idéia da prova. Construções mais complicadas requerem discussão adicional para provar corretude. Uma prova formal de corretude para

³Essa expressão definiria os estados de aceitação de M como sendo aqueles para os quais *ambos* os membros do par são estados de aceitação. Nesse caso M aceitaria a cadeia somente se ambos M_1 e M_2 o aceitassem, portanto a linguagem resultante seria a *interseção* e não a união. Na realidade, esse resultado prova que a classe de linguagens regulares é fechada sob interseção.

uma construção desse tipo usualmente procede por indução. Para um exemplo de uma construção provada correta, veja a prova do Teorema 1.28. A maior parte das construções que você encontrará neste curso são bastante simples e portanto não requerem uma prova formal de corretude.

Acabamos de mostrar que a união de duas linguagens regulares é regular, mostrando portanto que a classe das linguagens regulares é fechado sob a operação de união. Agora nos voltamos para a operação de concatenação e tentamos mostrar que a classe das linguagens regulares é fechada sob aquela operação também.

Teorema 1.12

A classe das linguagens regulares é fechada sob a operação de concatenação.

Em outras palavras, se A_1 e A_2 são linguagens regulares então $A_1 \circ A_2$ também o é.

Para provar esse teorema vamos tentar algo na linha da prova do caso da união. Como antes, podemos começar com autômatos finitos M_1 e M_2 reconhecendo as linguagens regulares A_1 e A_2 . Porém agora, ao invés de construir o autômato M para aceitar sua entrada se M_1 ou M_2 aceita, ele tem que aceitar se sua entrada pode ser quebrada em duas partes, onde M_1 aceita a primeira parte e M_2 aceita a segunda parte. O problema é que M não sabe onde quebrar sua entrada (i.e., onde a primeira parte termina e a segunda começa). Para resolver esse problema introduzimos uma técnica chamada não-determinismo.

1.2 Não-determinismo

Não-determinismo é um conceito útil que tem tido grande impacto sobre a teoria da computação. Até agora em nossa discussão, todo passo de uma computação segue de uma forma única do passo precedente. Quando uma máquina está num dado estado e lê o próximo símbolo de entrada, sabemos que estado será o próximo—ele está determinado. Chamamos isso de computação *determinística*. Em uma máquina **não-determinística**, várias escolhas podem existir para cada próximo estado em qualquer ponto.

Não-determinismo é uma generalização de determinismo, portanto todo autômato finito determinístico é automaticamente um autômato finito não-determinístico. Como a figura a seguir mostra, autômatos finitos não-determinísticos podem ter características adicionais.

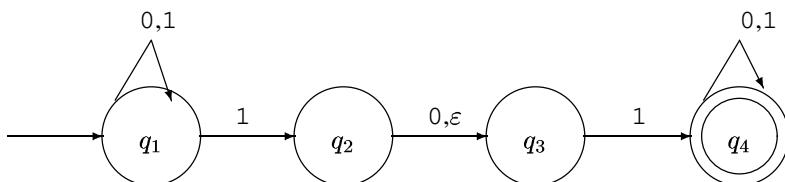


Figura 1.14: O autômato finito não-determinístico N_1

A diferença entre um autômato finito determinístico, abreviado por **AFD**, e um autômato finito não-determinístico, abreviado por **AFN**, é imediatamente aparente. Primeiro, todo estado de um **AFD** sempre tem exatamente uma seta de transição saindo para cada símbolo no alfabeto. O autômato não-determinístico mostrado na Figura 1.14 viola essa regra. O estado q_1 tem apenas uma seta saindo para o símbolo 0, mas tem duas para 1; q_2 tem uma seta para 0, mas não tem nenhuma para 1. Em um **AFN** um estado pode ter zero, um, ou mais setas saindo para cada símbolo do alfabeto.

Segundo, em um **AFD**, rótulos sobre as setas de transição são símbolos do alfabeto. Esse **AFN** tem uma seta com o rótulo ε . Em geral, um **AFN** pode ter setas rotuladas com membros do alfabeto ou com ε . Zero, uma, ou mais setas podem sair de cada estado com o rótulo ε .

Como um **AFN** computa? Suponha que você esteja rodando um **AFN** sobre uma cadeia de entrada e chegue num estado com múltiplas maneiras de prosseguir. Por exemplo, digamos que estamos no estado q_1 no **AFN** N_1 e que o próximo símbolo de entrada seja um 1. Após ler esse símbolo, a máquina divide-se em múltiplas cópias de si mesma e segue *todas* as possibilidades em paralelo. Cada cópia da máquina toma uma das possíveis maneiras de prosseguir e continua como antes. Se existirem escolhas subsequentes, a máquina divide-se novamente. Se o próximo símbolo de entrada não aparece sobre qualquer das setas saindo do estado ocupado por uma cópia da máquina, aquela cópia da máquina morre, juntamente com o ramo da computação associado a ela. Finalmente, se *qualquer uma* dessas cópias da máquina está em um estado de aceitação no final da entrada, o **AFN** aceita a cadeia de entrada.

Se um estado com um símbolo ε sobre uma seta saindo é encontrado, algo semelhante acontece. Sem ler qualquer entrada, a máquina divide-se em múltiplas cópias, uma seguindo cada uma das setas saindo rotuladas com ε e uma permanecendo no estado corrente. Então a máquina prossegue não-deterministicamente como antes.

Não-determinismo pode ser visto como uma espécie de computação paralela na qual vários “processos” podem estar rodando concorrentemente. Quando o **AFN** divide-se para seguir várias escolhas, isso corresponde a um processo “bifurcando” em vários filhos, cada um prosseguindo separadamente. Se pelo menos um desses processos aceita então a computação inteira aceita.

Uma outra maneira de pensar em uma computação não-determinística é como uma árvore de possibilidades. A raiz da árvore corresponde ao início da computação. Todo ponto de ramificação na árvore corresponde a um ponto na computação no qual a máquina tem múltiplas cópias. A máquina aceita se pelo menos um dos ramos da computação termina em um estado de aceitação, como mostrado na Figura 1.15.

Vamos considerar algumas rodadas amostrais do **AFN** N_1 mostrado na Figura 1.14. Sobre a entrada 010110 comece no estado inicial q_1 e leia o primeiro símbolo 0. A partir de q_1 existe somente um lugar para ir sobre 0, a saber, de volta a q_1 , portanto permaneça lá.

A seguir leia o segundo símbolo 1. Em q_1 sobre um 1 existem duas escolhas: ou permaneça em q_1 ou move para q_2 . Não-deterministicamente, a máquina divide-se em duas para seguir cada escolha. Mantenha o registro das possibilidades colocando um dedo sobre cada estado onde a máquina poderia estar. Portanto você agora tem dedos sobre os estados q_1 e q_2 . Uma seta ε sai do estado q_2 de modo que a máquina se divide novamente; mantenha um dedo sobre q_2 , e move o outro para q_3 . Você agora tem dedos sobre q_1 , q_2 , e q_3 .

Quando o terceiro símbolo 0 é lido, tome cada dedo por vez. Mantenha o dedo sobre q_1 no lugar, move o dedo sobre q_2 para q_3 , e remova o dedo que estava sobre q_3 .

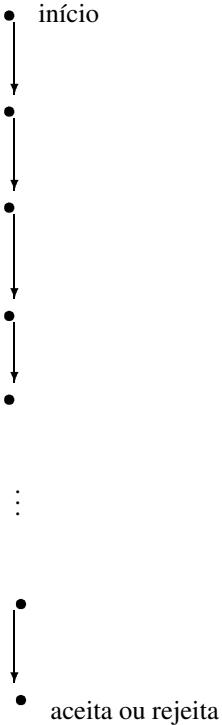
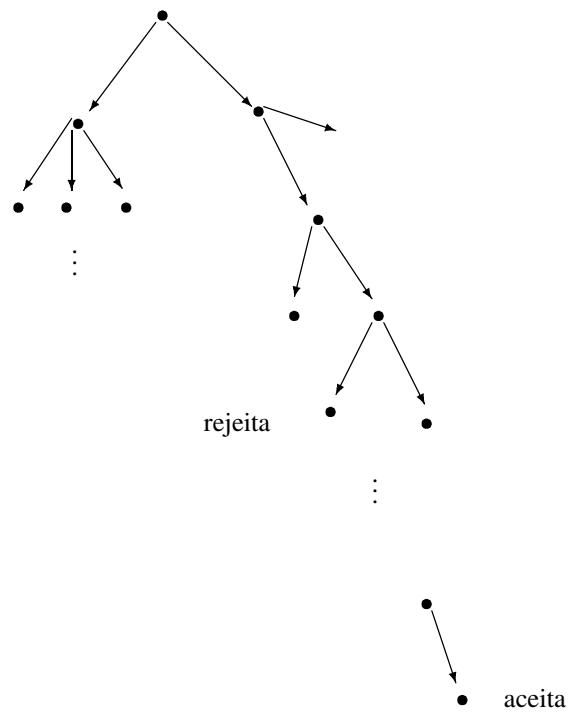
Computação
determinísticaComputação
não-determinística

Figura 1.15: Computações determinísticas e não-determinísticas com um ramo de aceitação

Esse último dedo não tinha nenhuma seta 0 para seguir e corresponde a um processo que simplesmente “morre.” Nesse ponto você tem dedos sobre os estados q_1 e q_3 .

Quando o quarto símbolo 1 é lido, divide o dedo sobre q_1 em dedos sobre q_1 e q_2 , e divide ainda o dedo sobre q_2 para seguir a seta ε para q_3 , e move o dedo que estava sobre q_3 para q_4 . Você agora tem um dedo sobre cada um dos quatro estados.

Quando o quinto símbolo 1 é lido, os dedos sobre q_1 e q_3 resultam em dedos sobre os estados q_1 , q_2 , q_3 , e q_4 , como você viu com o quarto símbolo. O dedo sobre o estado q_2 é removido. O dedo que estava sobre q_4 permanece sobre q_4 . Agora você tem dois dedos sobre q_4 , portanto remova um, pois você apenas precisa memorizar que q_4 é um estado possível nesse ponto, não que ele é possível por múltiplas razões.

Quando o sexto e último símbolo 0 é lido, mantenha o dedo sobre q_1 no lugar, move aquele sobre q_2 para q_3 , remove aquele que estava sobre q_3 , e deixe aquele sobre q_4 no lugar. Você está agora no final da cadeia, e você aceita se algum dedo está sobre um estado de aceitação. Você tem dedos sobre os estados q_1 , q_3 , e q_4 , e como q_4 é um estado de aceitação, N_1 aceita essa cadeia. A computação de N_1 sobre a entrada 010110 está ilustrada na Figura 1.16.

O que N_1 faz sobre a entrada 010? Comece com um dedo sobre q_1 . Após ler o 0 você ainda tem um dedo somente sobre q_1 , mas após o 1 existem dedos sobre q_1 , q_2 , e q_3 (não esqueça a seta ε). Após o terceiro símbolo 0, remove o dedo sobre q_3 , move

o dedo sobre q_2 para q_3 , e deixe o dedo sobre q_1 onde está. Nesse ponto você está no final da entrada, e como nenhum dedo está sobre um estado de aceitação, N_1 rejeita essa entrada.

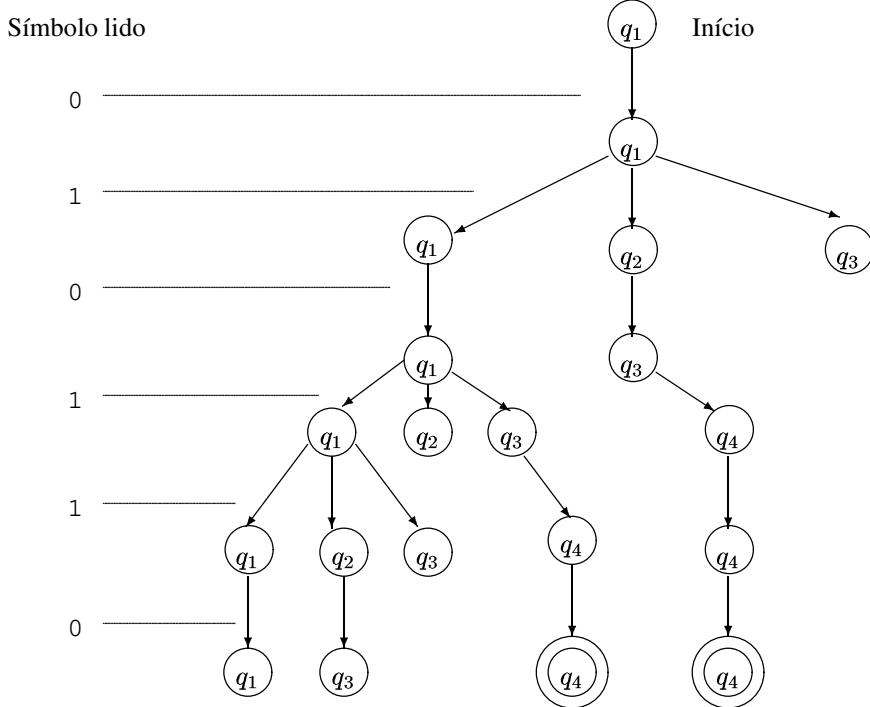


Figura 1.16: A computação de N_1 sobre a entrada 010110

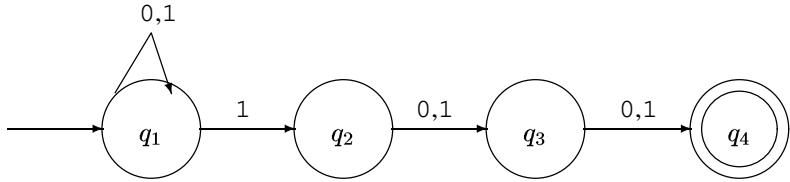
Continuando a experimentar dessa maneira, você verá que N_1 aceita todas as cadeias que contêm 101 ou 11 como subcadeia.

Autômatos finitos não-determinísticos são úteis em vários aspectos. Como mostraremos, todo AFN pode ser convertido em um AFD equivalente, e construir AFN's é às vezes mais fácil que construir diretamente AFD's. Um AFN pode ser muito menor que sua contrapartida determinística, ou seu funcionamento ser mais fácil de entender. Não-determinismo em autômatos finitos é também uma boa introdução a não-determinismo em modelos computacionais mais poderosos porque autômatos finitos são especialmente fáceis de entender. Agora nos voltamos para vários exemplos de AFN's.

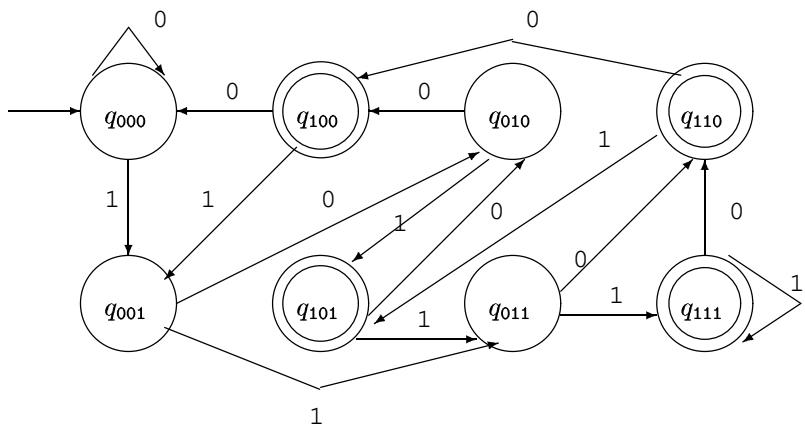
Exemplo 1.14

Seja A a linguagem consistindo de todas as cadeias sobre $\{0, 1\}$ contendo um 1 na terceira posição a partir do final (e.g., 000100 está em A mas 0011 não está). O seguinte AFN N_2 de quatro estados reconhece A .

Uma maneira boa de ver a computação desse AFN é dizer que ele permanece no estado inicial q_1 até que ele “adivinha” que ele está a três posições do final. Naquele ponto, se o símbolo de entrada é um 1, ele ramifica para o estado q_2 e usa q_3 e q_4 para “verificar” se seu palpite estava correto.

Figura 1.17: O AFN N_2 que reconhece A

Conforme mencionado anteriormente, todo AFN pode ser convertido em um AFD equivalente, mas às vezes aquele AFD pode ter muito mais estados. O menor AFD para A contém oito estados. Além disso, entender o funcionamento do AFD é muito mais fácil, como você pode ver examinando a Figura 1.18 para o AFD.

Figura 1.18: Um AFD que reconhece A

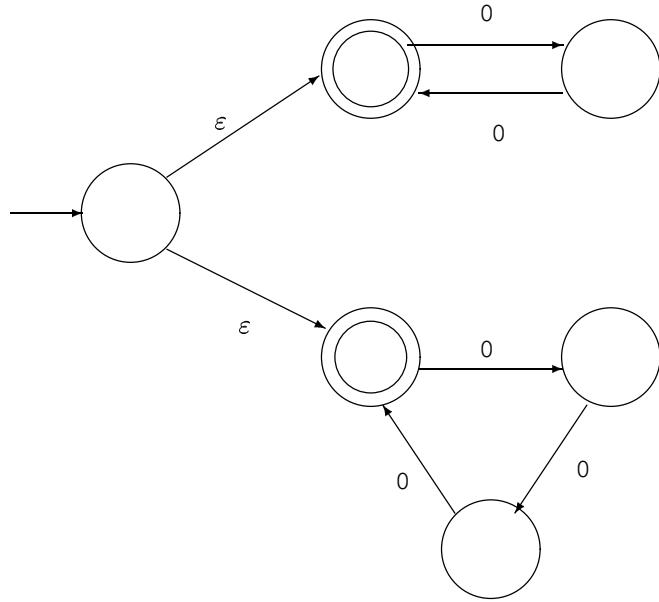
Suponha que adicionássemos ε aos rótulos sobre as setas indo de q_2 para q_3 e de q_3 para q_4 na máquina N_2 na Figura 1.17. Em outras palavras, ambas as setas teriam então o rótulo $0, 1, \varepsilon$ ao invés de $0, 1$. Que linguagem N_2 reconheceria com essa modificação? Tente modificar o AFD na Figura 1.18 para reconhecer essa linguagem.

Exemplo 1.15

Considere o seguinte AFN N_3 que tem como alfabeto de entrada $\{0\}$ consistindo de um único símbolo. Um alfabeto contendo apenas um símbolo é chamado de alfabeto *unário*.

Essa máquina demonstra a conveniência de se ter setas ε . Ela aceita cadeias da forma 0^k onde k é um múltiplo de 2 ou 3. (Lembre-se que o expoente denota repetição, e não exponenciação numérica.) Por exemplo, N_2 aceita as cadeias $\varepsilon, 00, 000, 0000$, e 000000 , mas não 0 ou 00000 .

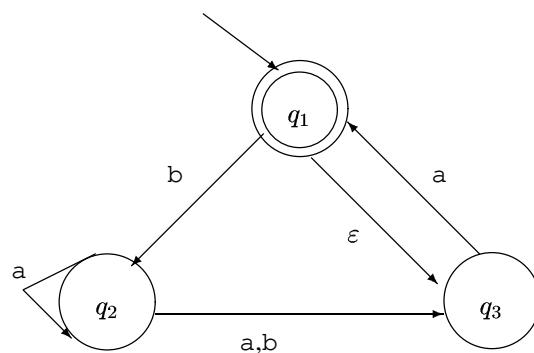
Pense na máquina operando de forma a inicialmente adivinhar se testa por um múltiplo de 2 ou um múltiplo de 3 ramificando ou no laço superior ou no laço inferior, e então verificando se seu palpite estava correto. É claro que poderíamos substituir essa

Figura 1.19: O AFN N_3

máquina por uma que não tem setas ϵ ou mesmo qualquer não-determinismo mesmo, mas a máquina mostrada é a mais fácil de entender essa linguagem.

Exemplo 1.16

Damos um outro exemplo de um AFN na figura seguinte. Pratique com ele para se satisfazer de que ele aceita as cadeias ϵ , a, baba, e baa, mas que ele não aceita as cadeias b, bb, e babba. Mas adiante usamos essa máquina para ilustrar o procedimento para converter AFN's em AFD's.

Figura 1.20: O AFN N_4

Definição formal de um autômato finito não-determinístico

A definição formal de um autômato finito não-determinístico é semelhante àquela de um autômato finito determinístico. Ambos têm estados, um alfabeto de entrada, uma função de transição, um estado inicial, e uma coleção de estados de aceitação. Entretanto, eles diferem de uma maneira essencial: no tipo de função de transição. Em um AFD a função de transição toma um estado e um símbolo de entrada e produz o próximo estado. Em um AFN a função de transição toma um estado e um símbolo de entrada *ou a cadeia vazia* e produz *o conjunto de estados seguintes possíveis*. De modo a escrever a definição formal, precisamos de fixar um pouco de notação adicional. Para qualquer conjunto Q escrevemos $\mathcal{P}(Q)$ como sendo a coleção de todos os subconjuntos de Q . Aqui $\mathcal{P}(Q)$ é chamado de *conjunto potência* de Q . Para qualquer alfabeto Σ escrevemos Σ_ϵ como sendo $\Sigma \cup \{\epsilon\}$. Agora podemos facilmente escrever a definição formal do tipo de função de transição em um AFN. Ela é: $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$, e estamos prontos para dar a definição formal.

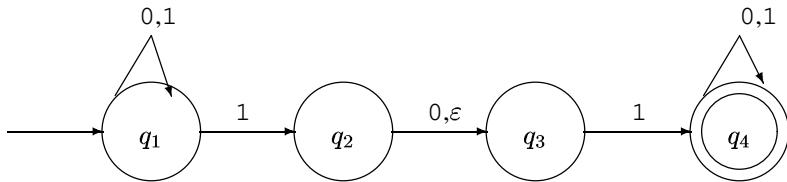
Definição 1.17

Um **autômato finito não-determinístico** é uma 5-upla $(Q, \Sigma, \delta, q_0, F)$ onde

1. Q é um conjunto finito de estados,
2. Σ é um alfabeto finito,
3. $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ é a **função de transição**,
4. $q_0 \in Q$ é o estado inicial, e
5. $F \subseteq Q$ é o conjunto de estados de aceitação.

Exemplo 1.18

Retomemos o autômato N_1 :



A descrição formal de N_1 é $(Q, \Sigma, \delta, q_1, F)$, onde

1. $q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. δ é dada como

	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	\emptyset
q_2	$\{q_3\}$	\emptyset	$\{q_3\}$
q_3	\emptyset	$\{q_4\}$	\emptyset
q_4	$\{q_4\}$	$\{q_4\}$	\emptyset

4. q_1 é o estado inicial, e
5. $F = \{q_4\}$.

A definição formal de computação de um AFN também é semelhante àquela para um AFD. Seja $N = (Q, \Sigma, \delta, q_0, F)$ um AFN e w uma cadeia sobre o alfabeto Σ . Então dizemos que N **aceita** w se podemos escrever w como $w = y_1 y_2 \cdots y_m$, onde cada y_i é um membro de Σ_ε e uma seqüência de estados r_0, r_1, \dots, r_m existe em Q com as seguintes três condições:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$ para $i = 0, \dots, m - 1$, e
3. $r_m \in F$.

Condição 1 diz que a máquina começa no estado inicial. Condição 2 diz que o estado r_{i+1} é um dos estados seguintes permissíveis quando N está no estado r_i e está lendo y_{i+1} . Observe que $\delta(r_i, y_{i+1})$ é o *conjunto* de estados seguintes permissíveis e portanto dizemos que r_{i+1} é um membro daquele conjunto. Finalmente, Condição 3 diz que a máquina aceita sua entrada se o último estado é um estado de aceitação.

Equivalência de AFN's e AFD's

Autômatos finitos determinísticos e não-determinísticos reconhecem a mesma classe de linguagens. Tal equivalência é tanto surpreendente quanto útil. Ela é surpreendente porque AFN's parecem ter mais poder que AFD's, portanto poderíamos esperar que AFN's reconhecessem mais linguagens. É útil porque descrever um AFN para uma dada linguagem às vezes é muito mais fácil que descrever um AFD para aquela linguagem.

Digamos que duas máquinas são *equivalentes* se elas reconhecem a mesma linguagem.

Teorema 1.19

Todo autômato finito não-determinístico tem um autômato finito determinístico equivalente.

Idéia da prova. Se uma linguagem é reconhecida por um AFN, então temos que mostrar a existência de um AFD que também a reconhece. A idéia é converter o AFN em um AFD equivalente que simula o AFN.

Lembre-se da estratégia “leitor como autômato” para projetar autômatos finitos. Como você simularia o AFN se você estivesse fazendo de conta que é um AFD? O que você precisaria memorizar à medida que a cadeia de entrada é processada? Nos exemplos de AFN's você memorizava as várias ramificações da computação colocando um dedo sobre cada estado que poderia estar ativo em dados pontos na entrada. Você atualizava os dedos movendo-, adicionando-, e removendo-os conforme a maneira com que o AFN opera. Tudo o que você precisava memorizar era o conjunto de estados com dedos sobre eles.

Se k é o número de estados do AFN, ele tem 2^k subconjuntos de estados. Cada subconjunto corresponde a uma das possibilidades de que o AFD tem que se lembrar, portanto o AFD que simula o AFN terá 2^k estados. Agora você precisa identificar

qual será o estado inicial e os estados de aceitação do AFD, e qual será sua função de transição. Podemos discutir isso mais facilmente fixando um pouco de notação formal.

Prova.

Seja $N = (Q, \Sigma, \delta, q_0, F)$ o AFN que reconhece uma dada linguagem A . Construimos um AFD M que reconhece A . Antes de realizar toda a construção, vamos primeiro considerar o caso mais fácil no qual N não tem setas ε . Mais adiante levaremos em conta as setas ε .

Construa $M = (Q', \Sigma, \delta', q_0', F')$.

1. $Q' = \mathcal{P}(Q)$.

Todo estado de M é um conjunto de estados de N . Lembre-se que $\mathcal{P}(Q)$ é o conjunto de todos os subconjuntos de Q .

2. Para $R \in Q'$ e $a \in \Sigma$ seja $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ para algum } r \in R\}$.

Se R é um estado de M , ele também é um conjunto de estados de N . Quando M lê um símbolo a no estado R , ele mostra para onde a leva cada estado em R . Devido ao fato de que cada estado pode ir para um conjunto de estados, tomamos a união de todos esses conjuntos. Uma outra forma de escrever essa expressão é

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).^4$$

3. $q_0' = \{q_0\}$.

M começa no estado correspondente à coleção contendo apenas o estado inicial de N .

4. $F' = \{R \in Q' \mid R \text{ contém um estado de aceitação de } N\}$.

A máquina M aceita se um dos estados possíveis em que N poderia estar nesse ponto é um estado de aceitação.

Agora precisamos considerar as setas ε . Para fazer isso fixemos um pouco mais de notação. Para qualquer estado R de M definimos $E(R)$ como sendo a coleção de estados que podem ser atingidos a partir de R percorrendo somente setas ε , incluindo os próprios membros de R . Formalmente, para $R \subseteq Q$ faça

$$E(R) = \{q \mid q \text{ pode ser atingido a partir de } R \text{ passando por 0 ou mais setas } \varepsilon\}.$$

Então modificamos a função de transição de M para colocar dedos a mais sobre todos os estados que podem ser atingidos passando por setas ε após cada passo. Substituindo $\delta(r, a)$ por $E(\delta(r, a))$ faz esse efeito. Por conseguinte

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ para algum } r \in R\}.$$

Adicionalmente precisamos modificar o estado inicial de M para mover os dedos inicialmente para todos os estados possíveis que podem ser atingidos a partir do estado inicial de N ao longo de setas ε . Mudando q_0' para $E(\{q_0\})$ faz esse efeito. Agora completamos a construção do AFD M que simula o AFN N .

⁴A notação $\bigcup_{r \in R} \delta(r, a)$ significa: a união dos conjuntos $\delta(r, a)$ para cada possível r em R .

A construção de M obviamente funciona corretamente. Em todos os passos da computação de M sobre uma entrada, ela claramente entra em um estado que corresponde ao subconjunto de estados em que N poderia estar naquele ponto. Por conseguinte nossa prova está completa.

Se a construção usada na prova anterior fosse mais complexa precisaríamos de provar que ela funciona tal qual reivindicado. Usualmente tais provas procedem por indução sobre o número de passos da computação. A maioria das construções que usamos neste livro são simples e portanto não requerem tal prova de corretude. Para ver um exemplo de uma construção mais complexa que provamos correta vá para a prova do Teorema 1.28.

O Teorema 1.19 enuncia que todo AFN pode ser convertido num AFD equivalente. Por conseguinte autômatos finitos não-determinísticos dão uma maneira alternativa de caracterizar linguagens regulares. Enunciamos esse fato como um corolário do Teorema 1.19.

Corolário 1.20

Uma linguagem é regular se e somente se algum autômato finito não-determinístico a reconhece.

Uma direção do “se e somente se” enuncia que uma linguagem é regular se algum AFN a reconhece. O Teorema 1.19 mostra que qualquer AFN pode ser convertido para um AFD equivalente, portanto se um AFN reconhece uma dada linguagem, o mesmo acontece com algum AFD, e portanto a linguagem é regular. A outra direção enuncia que uma linguagem é regular somente se algum AFN a reconhece. Ou seja, se uma linguagem é regular, algum AFN tem que a estar reconhecendo. Obviamente, essa condição é verdadeira pois uma linguagem regular tem um AFD que a reconhece e qualquer AFD é também um AFN.

Exemplo 1.21

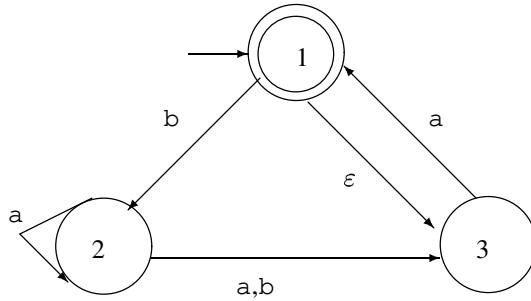
Vamos ilustrar o procedimento de converter um AFN para um AFD usando a máquina N_4 que foi dada no Exemplo 1.16. Para maior clareza, vamos renomear os estados de N_4 para $\{1, 2, 3\}$. Por conseguinte, a descrição formal de $N_4 = (Q, \{a, b\}, \delta, 1, \{1\})$, o conjunto de estados Q é $\{1, 2, 3\}$ como mostrado na Figura 1.21.

Para construir um AFD D que é equivalente a N_4 , primeiro determinamos os estados de D . N_4 tem três estados, $\{1, 2, 3\}$, portanto construímos D com oito estados, um para cada subconjunto dos estados de N_4 . Rotulamos cada um dos estados de D com o subconjunto correspondente. Por conseguinte o conjunto de estados de D é

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

A seguir, determinamos os estados inicial e de aceitação de D . O estado inicial é $E(\{1\})$, o conjunto de estados que são atingíveis a partir de 1 viajando por setas ε , mais o próprio 1. Uma seta ε vai de 1 para 3, portanto $E(\{1\}) = \{1, 3\}$. Os novos estados de aceitação são aqueles contendo o estado de aceitação de N_4 ; por conseguinte $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$.

Finalmente, determinamos a função de transição de D . Cada um dos estados de D vai para um lugar sobre a entrada a , e um lugar para a entrada b . Ilustramos o processo de determinar a colocação das setas de transição de D com uns poucos exemplos.

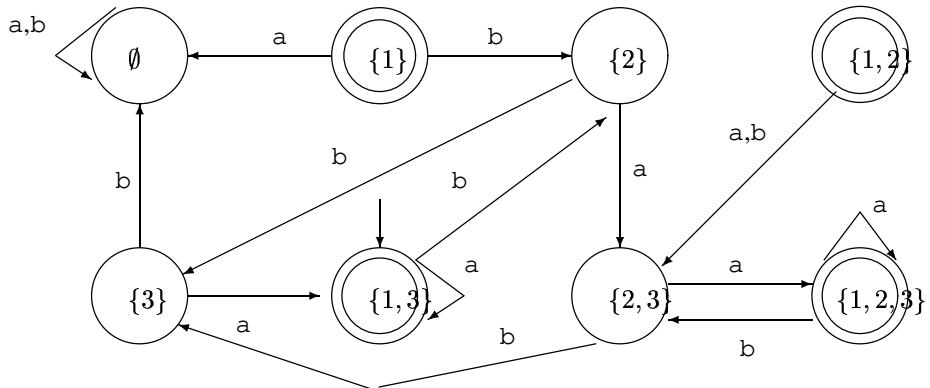

 Figura 1.21: O AFN N_4

Em D , o estado $\{2\}$ vai para $\{2, 3\}$ sobre a entrada a , porque em N_4 o estado 2 vai tanto para 2 quanto para 3 sobre a entrada a e não podemos ir além de 2 ou 3 ao longo de setas ε . O estado $\{2\}$ vai para o estado $\{3\}$ sobre a entrada b , porque em N_4 o estado 2 vai somente para o estado 3 sobre a entrada b e não podemos ir além de 3 ao longo de setas ε .

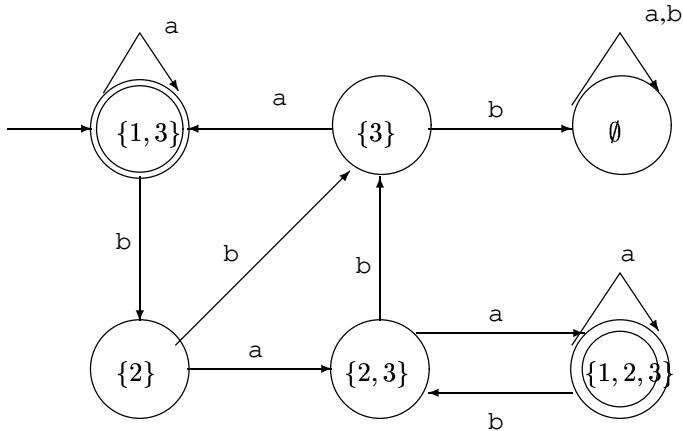
O estado $\{1\}$ vai para \emptyset sobre a , porque nenhuma seta a sai dele. Ele vai para $\{2\}$ sobre b .

O estado $\{3\}$ vai para $\{1, 3\}$ sobre a , porque em N_4 o estado 3 vai para 1 sobre a e 1 por sua vez vai para 3 com uma seta ε . O estado $\{3\}$ sobre b vai para \emptyset .

O estado $\{1, 2\}$ sobre a vai para $\{2, 3\}$ porque 1 não aponta para nenhum estado com setas a e 2 aponta tanto para 2 quanto para 3 com setas a e nenhum aponta para lugar nenhum com setas ε . O estado $\{1, 2\}$ sobre b vai para $\{2, 3\}$. Continuando dessa forma obtemos o seguinte diagrama para D .


 Figura 1.22: Um AFD D que é equivalente ao AFN N_4

Podemos simplificar essa máquina observando que nenhuma seta aponta para os estados $\{1\}$ e $\{1, 2\}$, portanto eles podem ser removidos sem afetar o desempenho da máquina. Fazendo isso obtém-se a seguinte figura:

Figura 1.23: O AFD D após a remoção de estados desnecessários

Fecho sob as operações regulares

Agora voltamos ao fecho da classe das linguagens regulares sob as operações regulares que começamos na Seção 1.1. Nosso objetivo é provar que a união, concatenação, e estrela de linguagens regulares são ainda regulares. Abandonamos a tentativa original de fazer isso quando lidar com a operação de concatenação era complicado demais. O uso de não-determinismo torna as provas mais fáceis.

Primeiro, vamos considerar novamente o fecho sob união. Anteriormente provamos o fecho sob união simulando deterministicamente ambas as máquinas simultaneamente via uma construção de produto cartesiano. Agora damos uma nova prova para ilustrar a técnica do não-determinismo. Revisar a primeira prova, na página ??, pode valer a pena para ver quão mais fácil e mais intuitiva a nova prova é.

Teorema 1.22

A classe de linguagens regulares é fechada sob a operação de união.

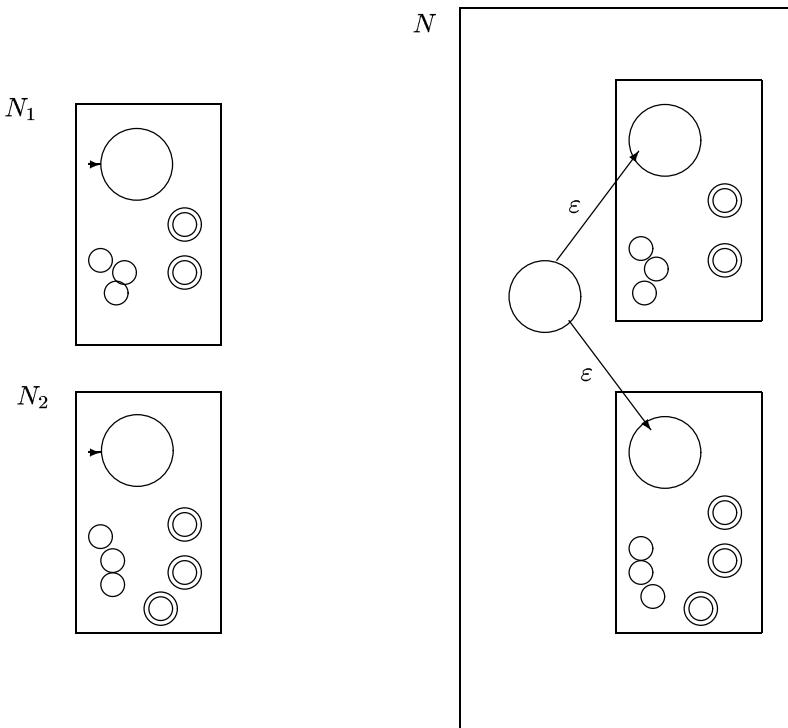
Idéia da prova. Temos linguagens regulares A_1 e A_2 e desejamos provar que $A_1 \cup A_2$ é regular. A idéia é tomar dois AFD's, N_1 e N_2 para A_1 e A_2 , e combiná-los em um novo AFD N .

A máquina N tem que aceitar sua entrada se N_1 ou N_2 aceita sua entrada. A nova máquina tem um novo estado inicial que ramifica para os estados iniciais das máquinas antigas com setas ε . Dessa maneira a nova máquina não-deterministicamente adivinha qual das duas máquinas aceita a entrada. Se uma delas aceita a entrada, N a aceitará também.

Representamos essa construção na Figura 1.24. À esquerda indicamos os estados inicial e de aceitação das máquinas N_1 e N_2 com círculos maiores e alguns estados adicionais com círculos menores. À direita mostramos como combinar N_1 e N_2 em N adicionando setas extra de transição.

Prova.

Seja $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, e

Figura 1.24: Construção de um AFN N para reconhecer $A_1 \cup A_2$

$$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2).$$

Construa $N = (Q, \Sigma, \delta, q_0, F)$ para reconhecer $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.

Os estados de N são todos estados de N_1 e N_2 , com a adição de um novo estado q_0 .

2. O estado q_0 é o estado inicial de N .

3. Os estados de aceitação $F = F_1 \cup F_2$.

Os estados de aceitação de N são todos os estados de aceitação de N_1 e N_2 . Dessa forma N aceita se N_1 aceita ou N_2 aceita.

4. Defina δ de modo que para qualquer $q \in Q$ e qualquer $a \in \Sigma_\varepsilon$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon \end{cases}$$

Agora podemos provar o fecho sob concatenação. Lembre-se que anteriormente, sem não-determinismo, completar a prova teria sido difícil.

Teorema 1.23

A classe de linguagens regulares é fechada sob a operação de concatenação.

Idéia da prova. Temos linguagens regulares A_1 e A_2 e desejamos provar que $A_1 \circ A_2$ é regular. A idéia é tomar dois AFN's N_1 e N_2 para A_1 e A_2 , e combiná-los em um novo AFN N como fizemos para o caso da união, mas dessa vez de uma forma diferente, como mostrado na Figura 1.25.

Associe o estado inicial de N ao estado inicial de N_1 . Os estados de aceitação de N_1 têm setas ϵ adicionais que não-deterministicamente permitem ramificação para N_2 sempre que N_1 está em um estado de aceitação, significando que ela encontrou uma parte inicial da cadeia que constitui uma cadeia em A_1 . Os estados de aceitação de N são estados de aceitação de N_2 somente. Por conseguinte ele aceita quando a entrada pode ser dividida em duas partes, a primeira aceita por N_1 e a segunda aceita por N_2 . Podemos pensar em N como adivinando não-deterministicamente onde fazer a divisão.

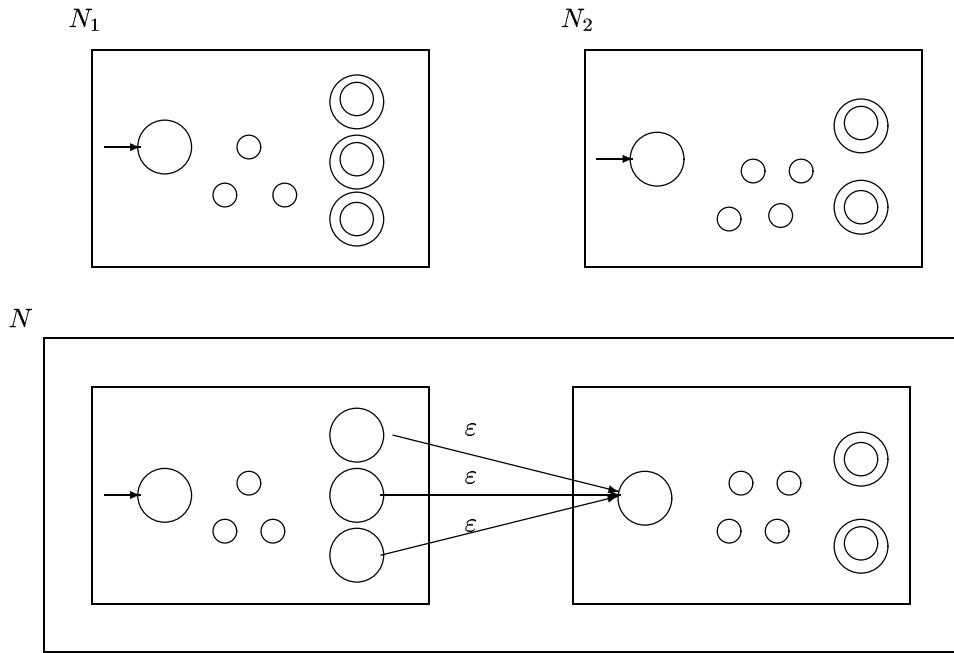


Figura 1.25: Construção de N para reconhecer $A_1 \circ A_2$

Prova.

Suponha que $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ reconhece A_1 , e

$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ reconhece A_2 .

Construa $N = (Q, \Sigma, \delta, q_1, F)$ para reconhecer $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.

Os estados de N são todos os estados de N_1 e N_2 .

2. O estado q_1 é o mesmo que o estado inicial de N_1 .

3. Os estados de aceitação F_2 são os mesmos que os estados de aceitação de N_2 .

4. Defina δ tal que para qualquer $q \in Q$ e qualquer $a \in \Sigma_\varepsilon$

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_2(q, a) & q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \text{ e } a = \varepsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

.....

Teorema 1.23

A classe de linguagens regulares é fechada sob a operação estrela.

Idéia da prova. Temos uma linguagem regular A_1 e queremos provar que A_1^* também é regular. Tomamos um AFN N_1 para A_1 e o modificamos para reconhecer A_1^* , como mostrado na Figura 1.26. O AFN resultante N aceitará sua entrada sempre que ela puder ser partida em vários pedaços em N_1 aceita cada pedaço.

Podemos construir N como N_1 com setas ε adicionais retornando ao estado inicial a partir do estado de aceitação. Dessa maneira, quando o processamento chega ao final de um pedaço que N_1 aceita, a máquina N tem a opção de pular para o estado inicial e tentar ler um outro pedaço que N_1 aceita. Adicionalmente temos que modificar N de tal modo que ela aceite ε , que sempre será um membro de A_1^* . Uma idéia (não muito boa) é simplesmente adicionar o estado inicial ao conjunto de estados de aceitação. Essa abordagem certamente adiciona ε à linguagem reconhecida, mas pode também adicionar outras cadeias indesejadas. O Exercício 1.11 pede um exemplo da falha dessa idéia. A forma de consertar esse problema é adicionar um novo estado inicial, que também é um estado de aceitação, e que tenha uma seta ε para o antigo estado inicial. Essa solução tem o efeito desejado de adicionar ε à linguagem sem adicionar qualquer outra coisa.

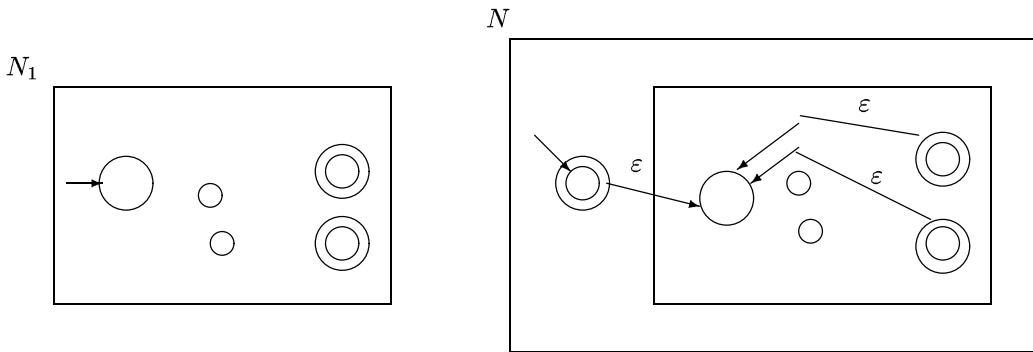


Figura 1.26: Construção de N para reconhecer A_1^*

Prova.

Suponha que $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ reconhece A_1 .

Construa $N = (Q, \Sigma, \delta, q_0, F)$ para reconhecer A_1^* .

$$1. Q = \{q_0\} \cup Q_1.$$

Os estados de N são os estados de N_1 mais um novo estado inicial.

$$2. O \text{ estado } q_0 \text{ é o novo estado inicial.}$$

$$3. F = \{q_0\} \cup F_1.$$

Os estados de aceitação são os antigos estados de aceitação mais o novo estado inicial.

$$4. Defina \delta \text{ tal que para qualquer } q \in Q \text{ e qualquer } a \in \Sigma_\varepsilon$$

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1 \text{ e } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon \end{cases}$$

1.3 Expressões regulares.....

Em aritmética, podemos usar as operações $+$ e \times para construir expressões tais como

$$(5 + 3) \times 4$$

Igualmente, podemos usar operações regulares para construir expressões descrevendo linguagens, que são chamadas *expressões regulares*. Um exemplo é

$$(0 \cup 1)^*$$

O valor da expressão aritmética é o número 32. O valor de uma expressão regular é uma linguagem. Nesse caso o valor é a linguagem consistindo de todas as cadeias com um 0 ou um 1 seguido de um número qualquer de 0's. Obtemos esse resultado dissecando a expressão em suas partes. Primeiro, os símbolos 0 e 1 são abreviações para os conjuntos $\{0\}$ e $\{1\}$. Portanto $(0 \cup 1)$ significa $(\{0\} \cup \{1\})$. O valor dessa parte é a linguagem $\{0, 1\}$. A parte 0^* significa $\{0\}^*$, e seu valor é a linguagem consistindo de todas as cadeias contendo um número qualquer de 0's. Segundo, como o símbolo \times em álgebra, o símbolo de concatenação \circ frequentemente está implícito nas expressões regulares. Por conseguinte $(0 \cup 1)0^*$ na verdade é uma abreviação para $(0 \cup 1) \circ 0^*$. A concatenação junta as cadeias das duas partes para obter o valor da expressão inteira.

Expressões regulares têm um papel importante em aplicações em ciência da computação. Em aplicações envolvendo texto, usuários podem querer buscar por cadeias que satisfazem certos padrões. Utilitários como AWK e GREP em UNIX, linguagens de programação como PERL, e editores de texto, todos provêem mecanismos para a descrição de padrões usando expressões regulares.

Exemplo 1.25

Um outro exemplo de uma expressão regular é

$$(0 \cup 1)^*$$

Ela começa com a linguagem $(0 \cup 1)$ e aplica a operação $*$. O valor dessa expressão é a linguagem consistindo de todas as cadeias possíveis de 0's e 1's. Se $\Sigma = \{0, 1\}$, podemos escrever Σ como abreviação para a expressão regular $(0 \cup 1)$. De um modo geral, se Σ é um alfabeto qualquer, a expressão regular Σ descreve a linguagem consistindo de todas as cadeias de comprimento 1 sobre esse alfabeto, e Σ^* descreve a linguagem consistindo de todas as cadeias sobre aquele alfabeto. Igualmente Σ^*1 é a linguagem que contém todas as cadeias que terminam em um 1. A linguagem $(0\Sigma^*) \cup (\Sigma^*1)$ consiste de todas as cadeias que ou começam com um 0 ou terminam com um 1.

Em aritmética, dizemos que \times tem precedência sobre $+$ para dizer que, quando existe uma escolha, fazemos a operação \times primeiro. Por conseguinte em $2 + 3 \times 4$ o 3×4 é realizado antes da adição. Para ter a adição realizada primeiro temos que adicionar parênteses para obter $(2 + 3) \times 4$. Em expressões regulares, a operação estrela é realizada primeiro, seguida por concatenação, e finalmente união, a menos que parênteses sejam usados para mudar a ordem usual.

Definição formal de uma expressão regular

Definição 1.26

Digamos que R é uma **expressão regular** se R é

1. a para algum a no alfabeto Σ ,
2. ε ,
3. \emptyset ,
4. $(R_1 \cup R_2)$, onde R_1 e R_2 são expressões regulares,
5. $(R_1 \circ R_2)$, onde R_1 e R_2 são expressões regulares, ou
6. (R_1^*) , onde R_1 é uma expressão regular.

Nos itens 1 e 2, as expressões regulares a e ε representam as linguagens $\{a\}$ e $\{\varepsilon\}$, respectivamente. No item 3, a expressão regular \emptyset representa a linguagem vazia. Nos itens 4,5, e 6, as expressões representam as linguagens obtidas tomando-se a união ou concatenação das linguagens R_1 e R_2 , ou a estrela da linguagem R_1 , respectivamente.

Não confunda as expressões regulares ε e \emptyset . A expressão ε representa a linguagem contendo uma única cadeia, a saber, a cadeia vazia, enquanto que \emptyset representa a linguagem que não contém qualquer cadeia.

Aparentemente, estamos sob risco de definir a noção de expressão regular em termos de si própria. Se verdadeiro, teríamos uma **definição circular**, o que seria inválido. Entretanto, R_1 e R_2 são sempre menores que R . Por conseguinte estamos na verdade definindo expressões regulares em termos de expressões regulares menores e dessa forma evitando circularidade. Uma definição desse tipo é chamada de **definição induutiva**.

Parênteses em uma expressão regular podem ser omitidos. Se assim for, o cálculo é feito na ordem de precedência: estrela, então concatenação, então união.

Quando desejamos deixar claro uma distinção entre uma expressão regular R e a linguagem que ela descreve, escrevemos $L(R)$ como sendo a linguagem de R .

Exemplo 1.27

Nos exemplos abaixo assumimos que o alfabeto Σ é $\{0, 1\}$.

1. $0^*10^* = \{w \mid w \text{ tem exatamente um único } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ tem pelo menos um } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contém a cadeia } 001 \text{ como uma subcadeia}\}$.
4. $(\Sigma\Sigma)^* = \{w \mid w \text{ é uma cadeia de comprimento par}\}.$ ⁵
5. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{o comprimento de } w \text{ é um múltiplo de três}\}$.
6. $01 \cup 10 = \{01, 10\}$.
7. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ começa e termina com o mesmo símbolo}\}$.
8. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.

A expressão $0 \cup \varepsilon$ descreve a linguagem $\{0, \varepsilon\}$, de modo que a operação de concatenação adiciona 0 ou ε antes de toda cadeia em 1^* .

9. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.

$$10. 1^*\emptyset = \emptyset.$$

Concatenar o conjunto vazio a um conjunto qualquer resulta no conjunto vazio.

$$11. \emptyset^* = \varepsilon.$$

A operação estrela junta qualquer número de cadeias da linguagem para obter uma cadeia no resultado. Se a linguagem é vazia, a operação estrela pode juntar 0 cadeias, dando apenas a cadeia vazia.

Se supormos que R é uma expressão regular qualquer, temos as seguintes identidades. Elas são bons testes para ver se você entende a definição.

$$R \cup \emptyset = R$$

Adicionar a linguagem vazia a qualquer outra linguagem não a modificará.

$$R \circ \varepsilon = R.$$

Adicionar a cadeia vazia a qualquer cadeia não a modificará.

Entretanto, trocando \emptyset por ε e vice-versa nas identidades acima pode fazer as igualdades falharem.

$$R \cup \varepsilon \text{ pode não ser igual a } R.$$

Por exemplo, se $R = 0$, então $L(R) = \{0\}$ mas $L(R \cup \varepsilon) = \{0, \varepsilon\}$.

$$R \circ \emptyset \text{ pode não ser igual a } R.$$

Por exemplo, se $R = \{0\}$, então $L(R) = \{0\}$ mas $L(R \circ \emptyset) = \emptyset$.

Expressões regulares são ferramentas úteis no desenho de compiladores para linguagens de programação. Objetos elementares em uma linguagem de programação, chamados de **tokens**, tais como nomes de variáveis e de constantes, podem ser descritos com expressões regulares. Por exemplo, uma constante numérica que pode incluir uma parte fracionária e/ou um sinal pode ser descrita como um membro da linguagem

$$(+ \cup - \cup \varepsilon)(DD^* \cup DD^*, D^* \cup D^*, DD^*),$$

⁵O *comprimento* de uma cadeia é o número de símbolos que ela contém.

onde $D = \{0, 1, 2, 3.4, 5, 6, 7, 8, 9\}$ é o alfabeto de dígitos decimais. Exemplos de cadeias geradas são: $72, 3, 14159, +7,,$ e $-01.$

Uma vez que a sintaxe dos tokens da linguagem de programação tenha sido descrita com expressões regulares, sistemas automáticos podem gerar o *analisador léxico*, a parte de um compilador que inicialmente processa o programa de entrada.

Equivalência com autômatos finitos

Expressões regulares e autômatos finitos são equivalentes no seu poder descritivo. Esse fato é um tanto notável, porque autômatos finitos e expressões regulares aparentam superficialmente ser um tanto diferentes. Entretanto, qualquer expressão regular pode ser convertida em um autômato finito que reconhece a linguagem que ela descreve, e vice-versa. Lembre-se que uma linguagem regular é aquela que é reconhecida por algum autômato finito.

Teorema 1.28

Uma linguagem é regular se e somente se alguma expressão regular a descreve.

Esse teorema tem duas direções. Enunciamos e provamos cada direção como um lema separado.

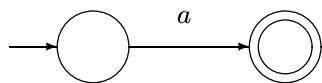
Lema 1.29

Se uma linguagem é descrita por uma expressão regular, então ela é regular.

Idéia da prova. Digamos que temos uma expressão regular R descrevendo uma certa linguagem A . Mostramos como converter R em um AFN que reconhece A . Pelo Corolário 1.20, se um AFN reconhece A então A é regular.

Prova. Vamos converter R em um AFN N . Consideraremos os seis casos na definição formal de expressões regulares.

1. $R = a$ para algum $a \in \Sigma$. Então $L(R) = \{a\}$, e o AFN abaixo reconhece $L(R)$.

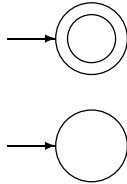


Note que essa máquina se encaixa na definição de um AFN mas não na de um AFD pois ela tem alguns estados sem seta saindo para cada símbolo de entrada possível. É claro que poderíamos ter apresentado aqui um AFD equivalente mas um AFN é tudo do que precisamos por agora, e é mais fácil descrever.

Formalmente, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, onde descrevemos δ dizendo que $\delta(q_1, a) = \{q_2\}$, $\delta(r, b) = \emptyset$ para $r \neq q_1$ ou $b \neq a$.

2. $R = \varepsilon$. Então $L(R) = \{\varepsilon\}$, e o seguinte AFN reconhece $L(R)$.

Formalmente, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, onde $\delta(r, b) = \emptyset$ para qualquer r e b .



3. $R = \emptyset$. Então $L(R) = \emptyset$, e o seguinte AFN reconhece $L(R)$.

Formalmente $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, onde $\delta(r, b) = \emptyset$ para qualquer r e b .

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

Para os três últimos casos usamos as construções dadas nas provas de que a classe das linguagens regulares é fechada sob as operações regulares. Em outras palavras, construimos o AFN para R a partir dos AFN's para R_1 e R_2 (ou somente R_1 no caso 6) e a construção de fecho apropriada.

.....

Isso conclui a primeira parte da prova do Teorema 1.28, dando a direção mais fácil do se e somente se. Antes de prosseguir para a outra direção vamos considerar alguns exemplos através dos quais usamos esse procedimento para converter uma expressão regular num AFN.

Exemplo 1.30

Convertemos a expressão regular $(ab \cup a)^*$ em um AFN em uma seqüência de estágios. Montamos a partir das subexpressões menores para as subexpressões maiores até que tenhamos um AFN para a expressão original, como mostrado no diagrama abaixo. Note que esse procedimento geralmente não dá o AFN com o mínimo de estados. Neste exemplo, o procedimento dá um AFN com oito estados, mas o menor AFN equivalente tem apenas dois estados. Você pode encontrá-lo?

Exemplo 1.31

Neste segundo exemplo convertemos a expressão regular $(a \cup b)^* aba$ em um AFN. (Veja Figura 1.28.) Alguns dos passos intermediários não são mostrados.

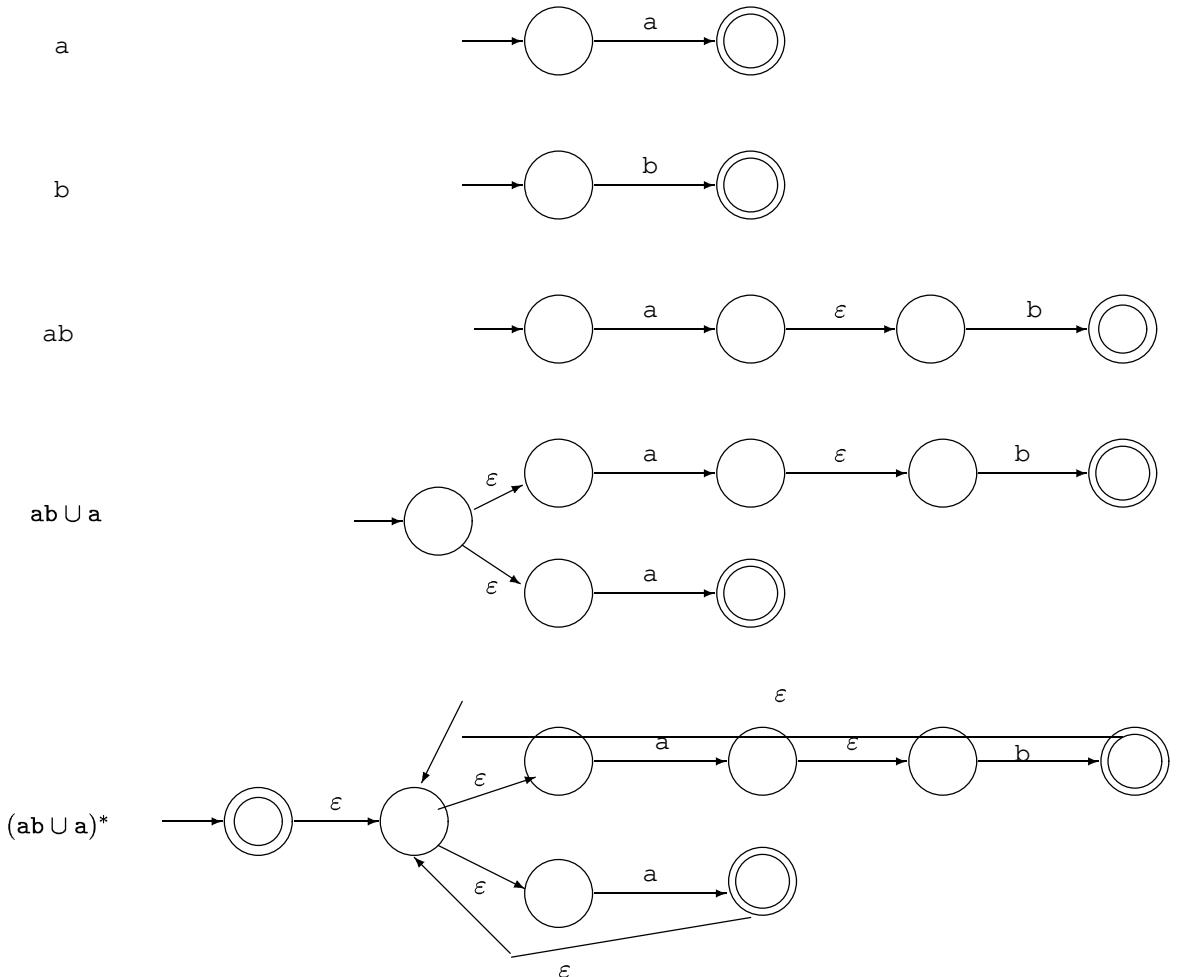
Agora vamos nos voltar para a outra direção da prova do Teorema 1.28.

Lema 1.32

Se uma linguagem é regular, então ela é descrita por uma expressão regular.

.....

Idéia da prova. Precisamos mostrar que, se uma linguagem A é regular, uma expressão regular a descreve. Devido ao fato de que A é regular, ela é aceita por um


 Figura 1.27: Construindo um AFN a partir da expressão regular $(ab \cup a)^*$

AFD. Descrevemos um procedimento para converter AFD's em expressões regulares equivalentes.

Dividimos esse procedimento em duas partes, usando um novo tipo de autômatos finitos chamado de **autômato finito não-determinístico generalizado**, AFNG. Primeiro mostramos como converter AFD's em AFNG's e daí AFNG's em expressões regulares.

Autômatos finitos não-determinísticos generalizados são simplesmente autômatos finitos não-determinísticos nos quais as setas de transição podem ter qualquer expressão regular como rótulo, ao invés de apenas membros do alfabeto ou ϵ . O AFNG lê blocos de símbolos da entrada, não necessariamente apenas um símbolo a cada vez como num AFD comum. O AFNG se move ao longo de uma seta de transição conectando dois estados após ler um bloco de símbolos da entrada, o qual constitui uma cadeia descrita pela expressão regular sobre aquela seta. Um AFNG é não-determinístico e portanto pode ter várias maneiras diferentes de processar a mesma cadeia de entrada. Ele aceita sua entrada se seu processamento pode levar o AFNG a estar em um estado de aceitação no final da entrada. A Figura 1.29 apresenta um exemplo de um AFNG.

Por conveniência requeremos que AFNG's sempre tenham um formato especial que atende às seguintes condições:

- O estado inicial tem setas de transição indo para todos os outros estados mas nenhuma chegando de nenhum outro estado.
- Existe apenas um único estado de aceitação, e ele tem setas vindo de todos os outros estados mas nenhuma seta saindo para nenhum outro estado. Além disso, o estado de aceitação não é o mesmo que o estado inicial.
- Exceto os estados inicial e de aceitação, uma seta vai de todo estado para todos os outros estados e de cada estado para si próprio.

Podemos facilmente converter um AFD em um AFNG no formato especial. Simplesmente adicionamos um novo estado inicial com uma seta ε ao antigo estado inicial e um novo estado de aceitação com setas ε a partir dos antigos estados de aceitação. Se quaisquer setas têm múltiplos rótulos (ou se existem múltiplas setas indo entre os mesmos dois estados na mesma direção), substituimos cada uma por uma única seta cujo rótulo é a união dos rótulos anteriores. Finalmente, adicionamos setas rotuladas com \emptyset entre estados que não tinham setas. Esse último passo não modificará a linguagem reconhecida porque uma transição rotulada com \emptyset nunca pode ser usada. Daqui por diante assumimos que todos os AFNG's estão no formato especial.

Agora vamos mostrar como converter um AFNG em uma expressão regular. Digamos que o AFNG tem k estados. Então, devido ao fato de que um AFNG tem que ter um estado inicial e um estado de aceitação e eles têm que ser diferentes um do outro, sabemos que $k \geq 2$. Se $k > 2$, construimos um AFNG equivalente com $k - 1$ estados. Esse passo pode ser repetido sobre o novo AFNG até ele ser reduzido a dois estados. Se $k = 2$, o AFNG tem uma única seta que vai do estado inicial para o estado de aceitação. O rótulo dessa seta é a expressão regular equivalente. Por exemplo, os estágios na conversão de um AFD com três estados para uma expressão regular equivalente são mostrados na Figura 1.30.

O passo crucial é na construção de um AFNG equivalente com um estado a menos quando $k > 2$. Fazemos isso selecionando um estado, removendo-o da máquina, e reparando o restante de modo que a mesma linguagem seja ainda reconhecida. Qualquer estado serve, desde que ele não seja o estado inicial ou o estado de aceitação. Temos garantido de que tal estado existirá pois $k > 2$. Vamos chamar esse estado removido q_{rem} .

Após remover q_{rem} reparamos a máquina alterando as expressões regulares que rotulam cada uma das setas remanescentes. Os novos rótulos compensam a ausência de q_{rem} adicionando de volta as computações perdidas. O novo rótulo indo de um estado q_i para um estado q_j é uma expressão regular que descreve todas as cadeias que levariam a máquina de q_i para q_j ou diretamente ou via q_{rem} . Ilustramos essa abordagem na Figura 1.31.

Na máquina antiga se q_i vai de q_{rem} com uma seta rotulada R_1 , q_{rem} vai para si próprio com uma seta rotulada R_2 , q_{rem} vai para q_j com uma seta rotulada R_3 , e q_i vai para q_j com uma seta rotulada R_4 , então na nova máquina a seta de q_i para q_j recebe o rótulo

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

Fazemos essa mudança para cada seta indo de qualquer estado q_i para qualquer estado q_j , incluindo o caso em que $q_i = q_j$. A nova máquina reconhece a linguagem original.

Prova. Vamos agora realizar essa idéia formalmente. Primeiro, para facilitar a prova, definimos formalmente o novo tipo de autômato introduzido. Um AFNG é semelhante a um autômato finito não-determinístico exceto pela função de transição, que tem a forma

$$\delta : (Q - \{q_{\text{aceita}}\}) \times (Q - \{q_{\text{inicio}}\}) \longrightarrow \mathcal{R}.$$

O símbolo \mathcal{R} é a coleção de todas as expressões regulares sobre o alfabeto Σ , e q_{aceita} e q_{inicio} são os estados inicial e de aceitação. Se $\delta(q_i, q_j) = R$, a seta do estado q_i para o estado q_j tem a expressão regular R como seu rótulo. O domínio da função de transição é $(Q - \{q_{\text{aceita}}\}) \times (Q - \{q_{\text{inicio}}\})$ porque uma seta conecta todo estado para todos os outros estados, exceto que nenhuma seta está vindo de q_{aceita} ou chegando em q_{inicio} .

Definição 1.33

Um *autômato finito não-determinístico generalizado*, $(Q, \Sigma, \delta, q_{\text{inicio}}, q_{\text{aceita}})$, é uma 5-upla onde

1. Q é o conjunto finito de estados,
2. Σ é o alfabeto de entrada,
3. $\delta : (Q - \{q_{\text{aceita}}\}) \times (Q - \{q_{\text{inicio}}\}) \longrightarrow \mathcal{R}$ é a função de transição,
4. q_{inicio} é o estado inicial, e
5. q_{aceita} é o estado de aceitação.

Um AFNG aceita uma cadeia w em Σ^* se $w = w_1 w_2 \cdots w_k$, onde cada w_i está em Σ^* e uma seqüência de estados q_0, q_1, \dots, q_k existe tal que

1. $q_0 = q_{\text{inicio}}$ é o estado inicial,
2. $q_k = q_{\text{aceita}}$ é o estado de aceitação, e
3. para cada i , temos $w_i \in L(R_i)$, onde $R_i = \delta(q_{i-1}, q_i)$; em outras palavras, R_i é a expressão sobre a seta de q_{i-1} para q_i .

Voltando à prova do Lema 1.32, supomos que M seja o AFD para a linguagem A . Então convertemos M para um AFNG G adicionando um novo estado inicial e um novo estado de aceitação e setas adicionais de transição conforme necessário. Usamos o procedimento *CONVERT*(G), que toma um AFNG como entrada e retorna uma expressão regular equivalente. Esse procedimento usa *recursão*, o que significa que ele chama a si próprio. Um laço infinito é evitado porque o procedimento chama a si próprio somente para processar o AFNG que tem um estado a menos. O caso em que o AFNG tem dois estados é tratado sem recursão.

CONVERT(G):

1. Seja k o número de estados de G .

2. Se $k = 2$, então G tem que consistir de um estado inicial, um estado final, e uma única seta conectando-os e rotulada com uma expressão regular R .

Retorne a expressão R .

3. Se $k > 2$, selecionamos qualquer estado $q_{\text{rem}} \in Q$ diferente de q_{inicio} e q_{aceita} , e montamos G' como sendo o AFNG $(Q', \Sigma, \delta', q_{\text{inicio}}, q_{\text{aceita}})$, onde

$$Q' = Q - \{q_{\text{rem}}\},$$

e para qualquer $q_i \in Q' - \{q_{\text{aceita}}\}$ e qualquer $q_j \in Q' - \{q_{\text{inicio}}\}$ faça

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$

para $R_1 = \delta(q_i, q_{\text{rem}})$, $R_2 = \delta(q_{\text{rem}}, q_{\text{rem}})$, $R_3 = \delta(q_{\text{rem}}, q_j)$, e $R_4 = \delta(q_i, q_j)$.

4. Calcule $\text{CONVERT}(G')$ e retorne esse valor.

Agora provamos que $\text{CONVERT}(G)$ retorna o valor correto.

Afirmiação 1.34

Para qualquer AFNG G , $\text{CONVERT}(G)$ é equivalente a G .

Provamos essa afirmação por indução sobre k , o número de estados do AFNG.

Base. Prove que a afirmação é verdadeira para $k = 2$ estados. Se G tem apenas dois estados, ele pode ter apenas uma única seta, que vai do estado inicial para o estado de aceitação. O rótulo sob forma de expressão regular sobre essa seta descreve todas as cadeias que permitem que G chegue ao estado de aceitação. Donde essa expressão é equivalente a G .

Passo indutivo. Assuma que a afirmação é verdadeira para $k - 1$ estados e use essa suposição para provar que a afirmação é verdadeira para k estados. Primeiro mostramos que G e G' reconhecem a mesma linguagem. Suponha que G aceita uma entrada w . Então em um ramo de aceitação da computação G entra numa seqüência de estados

$$q_{\text{inicio}}, q_1, q_2, q_3, \dots, q_{\text{aceita}}.$$

Se nenhum deles é o estado removido q_{rem} , claramente G' também aceita w . A razão é que cada uma das novas expressões regulares rotulando as setas de G' contém a expressão regular antiga como parte de uma união.

Se q_{rem} realmente aparece, removendo cada rodada e estados q_{rem} consecutivos forma uma computação de aceitação para G' . Os estados q_i e q_j iniciando e terminando uma rodada têm uma nova expressão regular sobre a seta entre eles que descreve todas as cadeias levando q_i para q_j via q_{rem} sobre G . Portanto G' aceita w .

Para a outra direção, suponha que G' aceita uma cadeia w . Como cada seta entre quaisquer dois estados q_i e q_j em G' descreve a coleção de cadeias levando q_i para q_j em G , ou diretamente ou via q_{rem} , G tem que aceitar w também. Por conseguinte G e G' são equivalentes.

A hipótese de indução enuncia que quando o algoritmo chama a si próprio recursivamente sobre a entrada G' , o resultado é uma expressão regular que é equivalente a G' porque G' tem $k - 1$ estados. Daí a expressão regular também é equivalente a G , e o algoritmo está provado correto.

Isso conclui a prova da Afirmiação 1.34, Lema 1.32, e Teorema 1.28.

.....

Exemplo 1.35

Neste exemplo usamos o algoritmo precedente para converter um AFD em uma expressão regular. Começamos com o AFD de dois-estados na Figura 1.32(a).

Em (b) montamos um AFNG de quatro-estados adicionando um novo estado inicial e um novo estado de aceitação, chamados i e a ao invés de $q_{\text{início}}$ e q_{aceita} de modo que podemos desenhá-los convenientemente. Para evitar carregar demasiadamente a figura, não desenhamos as setas que são rotuladas \emptyset , muito embora elas estejam presentes. Note que substituímos o rótulo a, b sobre o auto-laço no estado 2 do AFD pelo rótulo $a \cup b$ no ponto correspondente sobre o AFNG. Fazemos isso porque o rótulo do AFD representa duas transições, uma para a e outra para b , enquanto que o AFNG pode ter somente uma única transição saindo de 2 para si próprio.

Em (c) removemos o estado 2, e atualizamos os rótulos de setas remanescentes. Neste caso, o único rótulo que muda é aquele de 1 para a . Em (b) era \emptyset , mas em (c) é $b(a \cup b)^*$. Obtemos esse resultado seguindo o passo 3 do procedimento *CONVERT*. O estado q_i é o estado 1, o estado q_j é a , e q_{rem} é 2, portanto $R_1 = b$, $R_2 = a \cup b$, $R_3 = \varepsilon$, e $R_4 = \emptyset$. Por conseguinte o novo rótulo sobre a seta de 1 para a é $(b)(a \cup b)^*(\varepsilon) \cup \emptyset$. Simplificamos essa expressão regular para $b(a \cup b)^*$.

Em (d) removemos o estado 1 de (c) e seguimos o mesmo procedimento. Devido ao fato de que somente os estados inicial e de aceitação permanecem, o rótulo sobre a seta que os conecta é a expressão regular que é equivalente ao AFD original.

Exemplo 1.36

Neste exemplo começamos com um AFD de três-estados. Os passos na conversão aparecem na Figura 1.33.

1.4 Linguagens não-regulares

Para entender o poder de autômatos finitos você também tem que entender suas limitações. Nesta seção mostramos como provar que certas linguagens não podem ser reconhecidas por nenhum autômato finito.

Vamos tomar a linguagem $B = \{0^n 1^n \mid n \geq 0\}$. Se tentarmos contrar um AFD que reconhece B , descobrimos que a máquina parece necessitar de memorizar quantos 0's foram vistos até então à medida que ela lê a entrada. Devido ao fato de que o número de 0's não é limitado, a máquina terá que manter registro de um número ilimitado de possibilidades. Mas ela não pode fazer com um nenhuma quantidade finita de estados.

A seguir apresentamos um método para provar que linguagens como B não são regulares. O argumento que acaba de ser dado não já prova não-regularidade, pois o número de 0's é ilimitado? Não, não prova. Simplesmente porque a linguagem parece requerer uma quantidade ilimitada de memória não significa que ela necessariamente é assim. Acontece de ser verdadeiro para a linguagem B , mas outras linguagens parecem requerer um número ilimitado de possibilidades, e ainda assim na verdade são regulares. Por exemplo, considere duas linguagens sobre o alfabeto $\Sigma = \{0, 1\}$:

$$C = \{w \mid w \text{ tem o mesmo número de } 0's \text{ e } 1's\}, \text{ e}$$

$$D = \{w \mid w \text{ tem o mesmo número de ocorrências de } 01 \text{ e } 10 \text{ como subcadeias}\}.$$

À primeira vista uma máquina reconhecedora aparenta necessitar contar em cada caso, e por conseguinte nenhuma linguagem aparenta ser regular. Como esperado, C não é regular, mas surpreendentemente D é regular!¹⁶ Por conseguinte nossa intuição pode às vezes nos levar para fora dos trilhos, motivo pelo qual precisamos de provas matemáticas para ter certeza. Nesta seção mostramos como provar que certas linguagens não são regulares.

O lema do bombeamento para linguagens regulares

Nossa técnica para provar não-regularidade provém de um teorema sobre linguagens regulares, tradicionalmente chamado **lema do bombeamento**. Esse teorema enuncia que todas as linguagens regulares têm uma propriedade especial. Se pudermos mostrar que uma linguagem não tem essa propriedade, estamos garantidos de que ela não é regular. A propriedade enuncia que todas as cadeias na linguagem podem ser “bombeadas” se elas são no mínimo tão longas quanto um certo valor, chamado o **comprimento de bombeamento**. Isso significa que cada cadeia dessa contém uma parte que pode ser repetida um número qualquer de vezes com a cadeia resultante permanecendo na linguagem.

Teorema 1.37

Lema do bombeamento Se A é uma linguagem regular, então existe um número p (o comprimento de bombeamento) onde, se s é uma cadeia qualquer de A de comprimento pelo menos p , então s pode ser dividida em três partes, $s = xyz$, satisfazendo as seguintes condições:

1. para cada $i \geq 0$, $xy^i z \in A$,
2. $|y| > 0$, e
3. $|xy| \leq p$.

Lembre-se da notação onde $|s|$ representa o comprimento da cadeia s , y^i significa que i cópias de y são concatenadas umas às outras, e y^0 é igual a ε .

Quando s é dividida em xyz , ou x ou z pode ser ε , mas a condição 2 diz que $y \neq \varepsilon$. Observe que sem essa condição 2 o teorema seria trivialmente verdadeiro. A condição 3 enuncia que as partes x e y juntas têm comprimento no máximo p . É uma condição técnica extra que ocasionalmente achamos útil ao provar que certas linguagens não são regulares. Veja o Exemplo 1.39 para uma aplicação da condição 3.

Idéia da prova. Seja $M = (Q, \Sigma, \delta, q_1, F)$ um AFD que reconhece A . Atribuimos o comprimento de bombeamento p como sendo o número de estados de M . Mostramos que qualquer cadeia s em A de comprimento pelo menos p pode ser quebrada nas três partes xyz satisfazendo nossas três condições. E se nenhuma cadeia em A é de comprimento pelo menos p ? Então nossa tarefa é ainda mais fácil porque o teorema se torna *vacuamente* verdadeiro: obviamente as três condições se verificam para todas as cadeias de comprimento pelo menos p se não existem tais cadeias.

Se s em A tem comprimento pelo menos p , considere a seqüência de estados pelos quais M passa quando computando com a entrada s . Ele começa com o estado inicial q_1 , e então vai para, digamos, q_3 , e daí para, digamos, q_{20} , e então para q_9 , e assim por

¹⁶Veja o Problema 1.41.

diante, até que ele atinge o final de s no estado q_{13} . Com s em A , sabemos que M aceita s , portanto q_{13} é um estado de aceitação.

Se dissermos que n é o comprimento de s , a seqüência de estados $q_1, q_3, q_{20}, q_9, \dots, q_{13}$ tem comprimento $n+1$. Devido ao fato de que n é pelo menos p , sabemos que $n+1$ é maior que p , o número de estados de M . Por conseguinte a seqüência tem que conter um estado repetido. Esse resultado é um exemplo do **princípio da casa-de-pombos**, um nome pomposo para o fato um tanto óbvio de que se p pombos são colocados em menos que p casas, alguma casa tem que ter mais que um pombo.

A Figura 1.34 mostra a cadeia s e a seqüência de estados pelos quais M passa quando processando s . O estado q_9 é um que se repete.

Agora dividimos s em três partes x , y , e z . A parte x é a parte de s que aparece antes de q_9 , a parte y é a parte entre as duas aparições de q_9 , e a parte z é a parte remanescente de s , vindo após a segunda ocorrência de q_9 . Portanto x leva M do estado q_1 para q_9 , y leva M de q_9 de volta para q_9 e z leva M de q_9 para o estado de aceitação q_{13} , como mostrado na Figura 1.35.

Vamos ver por que essa divisão satisfaz as três condições. Suponha que rodemos M sobre a entrada $xyyz$. Sabemos que x leva M de q_1 para q_9 , e então o primeiro y o leva de q_9 de volta a q_9 , o mesmo fazendo o segundo y , e então z o leva para q_{13} . Com q_{13} sendo um estado de aceitação, M aceita a cadeia $xyyz$. Igualmente, ele aceitará xy^iz para qualquer $i > 0$. Para o caso $i = 0$, $xy^iz = xz$, que é aceita por razões semelhantes. Isso estabelece a condição 1.

Verificando a condição 2, vemos que $|y| > 0$, pois era a parte de s que ocorria entre duas ocorrências diferentes do estado q_9 .

De modo a obter a condição 3, asseguramos que q_9 seja a primeira repetição na seqüência. Pelo princípio da casa-de-pombos, os primeiros $p+1$ estados na seqüência têm que conter uma repetição. Por conseguinte $|xy| \leq p$.

Prova. Seja $M = (Q, \Sigma, \delta, q_1, F)$ um AFD que reconhece A e p o número de estados de M .

Seja $s = s_1s_2\dots s_n$ uma cadeia em A de comprimento n , onde $n \geq p$. Seja r_1, \dots, r_{n+1} a seqüência de estados nos quais M entra enquanto processa s , portanto $r_{i+1} = \delta(r_i, s_i)$ para $1 \leq i \leq n$. Essa seqüência tem comprimento $n+1$, o que é pelo menos $p+1$. Entre os primeiros $p+1$ elementos na seqüência, dois têm que ser o mesmo estado, pelo princípio da casa-de-pombos. Chamamos o primeiro desses de r_j e o segundo de r_l . Devido ao fato de que r_l ocorre entre as primeiras $p+1$ posições na seqüência começando com r_1 , temos $l \leq p+1$. Agora faça $x = s_1\dots s_{j-1}$, $y = s_j\dots s_{l-1}$, e $z = s_l\dots s_n$.

Como x leva M de r_1 para r_j , y leva M de r_j para r_j , e z leva M de r_j para r_{n+1} , que é um estado de aceitação, M tem que aceitar xy^iz para $i \geq 0$. Sabemos que $j \neq l$, portanto $|y| > 0$; e $l \leq p+1$, logo $|xy| \leq p$. Por conseguinte satisfizemos todas as condições do lema do bombeamento.

Para usar o lema do bombeamento para provar que uma linguagem B não é regular, primeiro assuma que B é regular de modo a obter uma contradição. Então use o lema do bombeamento para garantir a existência de um comprimento de bombeamento p tal

que todas as cadeias de comprimento p ou mais em B podem ser bombeadas. A seguir, encontre uma cadeia s em B que tem comprimento p ou mais porém que não pode ser bombeada. Finalmente, demonstre que s não pode ser bombeada considerando todas as maneiras de se dividir x em x, y , e z (levando a condição 3 do lema do bombeamento em consideração se conveniente) e, para cada tal divisão, encontrando um valor i onde $xy^i z \notin B$. Esse passo final frequentemente envolve agrupar as várias maneiras de se dividir s em vários outros casos e analisá-los individualmente. A existência de s contradiz o lema do bombeamento se B fosse regular. De onde B não pode ser regular.

Encontrar s às vezes requer um pouco de pensamento criativo. Você pode precisar tentar vários candidatos para s antes de você descobrir um que funciona. Tente membros de B que parecem exibir a “essência” da não-regularidade de B . Discutimos mais a tarefa de encontrar s em alguns dos exemplos a seguir.

Exemplo 1.38

Seja B a linguagem $\{0^n 1^n \mid n \geq 0\}$. Usamos o lema do bombeamento para provar que B não é regular. A prova é por contradição.

Assuma, ao contrário, que B é regular. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Escolha s como sendo a cadeia $0^p 1^p$. Devido ao fato de que s é um membro de B e que s tem comprimento maior que p , o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, onde para qualquer $i \geq 0$ a cadeia $xy^i z$ está em B . Consideraremos três casos para mostrar que esse resultado é impossível.

1. A cadeia y consiste de apenas 0's. Neste caso a cadeia xyz tem mais 0's que 1's e portanto não é um membro de B , violando a condição 1 do lema do bombeamento. Este caso é uma contradição.
2. A cadeia y consiste de somente 1's. Este caso também dá uma contradição.
3. A cadeia y consiste de 0's e 1's. Neste caso a cadeia $xyyz$ pode ter o mesmo número de 0's e 1's, mas eles estarão fora de ordem com alguns 1's antes de 0's. Daí ela não é um membro de B , o que é uma contradição.

Por conseguinte uma contradição é inevitável se tomamos a suposição de que B é regular, portanto B não é regular.

Neste exemplo, encontrar a cadeia s foi fácil, porque qualquer cadeia em B de comprimento p ou mais funcionaria. Nos próximos dois exemplos algumas escolhas para s não funcionam, portanto cuidado adicional é necessário.

Exemplo 1.39

Seja $C = \{w \mid w \text{ tem o mesmo número de 0's e 1's}\}$. Usamos o lema do bombeamento para provar que C não é regular. A prova é por contradição.

Assuma, ao contrário, que C é regular. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Como no Exemplo 1.38, seja s a cadeia $0^p 1^p$. Com s sendo um membro de C e tendo comprimento maior que p , o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, onde para qualquer $i \geq 0$ a cadeia $xy^i z$ está em C . Gostaríamos de mostrar que esse resultado é impossível. Mas espere, ele é possível! Se fizermos x e z serem a cadeia vazia e y ser a cadeia $0^p 1^p$, então $xy^i z$ sempre terá o mesmo número de 0's e de 1's e portanto está em C . Portanto parece que s pode ser bombeada.

Aqui a condição 3 no lema do bombeamento é útil. Ela estipula que quando bombear s ela tem que ser dividida de tal modo que $|xy|p$. Essa restrição na forma em que s pode ser dividida torna mais fácil mostrar que a cadeia $s = 0^p 1^p$ que selecionamos não pode ser bombeada. Se $|xy| \leq p$, então y tem que consistir somente de 0's, portanto $xyyz \notin C$. Por conseguinte s não pode ser bombeada. Isso nos dá a contradição desejada.⁷

Selecionar a cadeia s neste exemplo exigiu mais cuidado do que no Exemplo 1.38. Se, ao contrário, tivéssemos escolhido $s = (01)^p$, teríamos esbarrado em problemas pois precisamos de uma cadeia que *não pode* ser bombeada e essa cadeia *pode* ser bombeada, mesmo levando em conta a condição 3. Você pode ver como bombeá-la? Uma maneira de fazê-lo é $x = \epsilon$, $y = 01$, e $z = (01)^{p-1}$. Então $xy^i z \in C$ para todo valor de i . Se você falhar na sua primeira tentativa de encontrar uma cadeia que não pode ser bombeada, não se desespere. Tente outra!

Um método alternativo de provar que C é não-regular segue de nosso conhecimento de que B é não-regular. Se C fosse regular, $C \cap 0^* 1^*$ também seria regular. As razões são que a linguagem $0^* 1^*$ é regular e que a classe das linguagens regulares é fechada sob interseção (provado na nota de pé de página na página ??). Mas $C \cap 0^* 1^*$ é igual a B , e sabemos que B é não-regular do Exemplo 1.38.

Exemplo 1.40

Seja $F = \{ww \mid w \in \{0, 1\}^*\}$. Mostramos que F é não-regular usando o lema do bombeamento.

Assuma, ao contrário, que F é regular. Seja p o comprimento do lema do bombeamento dado pelo lema do bombeamento. Seja s a cadeia $0^p 0^p 1$. Devido ao fato de que s é um membro de F e s tem comprimento maior que p , o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, satisfazendo as três condições do lema. Mostramos que esse resultado é impossível.

A condição 3 é novamente crucial, pois sem ela poderíamos bombear s se fizéssemos x e z ser a cadeia vazia. Com a condição 3 a prova segue porque y tem que consistir somente de 0's, portanto $xyyz \notin F$.

Observe que escolhemos $s = 0^p 1^p 1$ como sendo uma cadeia que exibe a “essência” da não-regularidade de F , ao invés de, digamos, a cadeia $0^p 0^p$. Muito embora $0^p 0^p$ seja um membro de F , ela não serve para demonstrar uma contradição porque ela pode ser bombeada.

Exemplo 1.41

Aqui demonstramos uma linguagem unária não-regular. Seja $D = \{1^{n^2} \mid n \geq 0\}$. Em outras palavras, D contém todas as cadeias de 1's cujo comprimento é um quadrado perfeito. Usamos o lema do bombeamento para provar que D não é regular. A prova é por contradição.

Assuma, ao contrário, que D é regular. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Seja s a cadeia 1^{p^2} . Devido ao fato de que s é um membro de D e s tem comprimento pelo menos p , o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, onde para qualquer $i \geq 0$ a cadeia $xy^i z$ está em D . Como nos exemplos precedentes, mostramos que esse resultado é impossível. Fazer isso neste caso requer um pouco mais de raciocínio com a seqüência de quadrados perfeitos:

$$0, 1, 4, 9, 16, 25, 36, 49, \dots$$

⁷Poderíamos ter usado a condição 3 no Exemplo 1.38 também, para simplificar sua prova.

Note a crescente distância entre membros sucessivos dessa seqüência. Membros grandes dessa seqüência não podem estar próximos um do outro.

Agora considere as duas cadeias xy^iz e $xy^{i+1}z$. Essas cadeias diferem entre si de uma única repetição de y , e consequentemente seus comprimentos diferem entre si do comprimento de y . Se escolhermos i muito grande, os comprimentos de xy^iz e $xy^{i+1}z$ não podem ser ambos quadrados perfeitos porque eles estão próximos demais. Por conseguinte xy^iz e $xy^{i+1}z$ não pode estar ambos em D , uma contradição.

Para tornar essa idéia uma prova, calculamos um valor de i que dê a contradição. Se $m = n^2$ é um quadrado perfeito, a diferença entre ele e o próximo quadrado perfeito $(n + 1)^2$ é

$$\begin{aligned}(n + 1)^2 - n^2 &= n^2 + 2n + 1 - n^2 \\ &= 2n + 1 \\ &= 2\sqrt{m} + 1\end{aligned}$$

O lema do bombeamento enuncia que ambos $|xy^iz|$ e $|xy^{i+1}z|$ são quadrados perfeitos para qualquer i . Mas, fazendo $|xy^iz|$ ser m como acima, vemos que eles *não podem* ser ambos quadrados perfeitos se $|y| < 2\sqrt{|xy^iz|} + 1$ porque eles estariam próximos demais.

Calculando o valor para i que leva a uma contradição agora é fácil. Observe que $|y| \leq |s| = p^2$. Seja $i = p^4$; então

$$\begin{aligned}|y| \leq p^2 &= \sqrt{p^4} \\ &< 2\sqrt{p^4} + 1 \\ &\leq 2\sqrt{|xy^iz|} + 1.\end{aligned}$$

Exemplo 1.41

Às vezes “bombear para baixo” é útil quando aplicamos o lema do bombeamento. Usamos o lema do bombeamento para mostrar que $E = \{0^i1^j \mid i > j\}$ não é regular. A prova é por contradição.

Assuma que E é regular. Seja p o comprimento de bombeamento para E dado pelo lema do bombeamento. Seja $s = 0^{p+1}1^p$. Então s pode ser dividida em xyz , satisfazendo as condições do lema do bombeamento. Pela condição 3, y consiste de apenas 0’s. Vamos examinar a cadeia $xyyz$ para ver se ela pode estar em E . Adicionando uma cópia extra de y aumenta o número de 0’s. Mas, E contém todas as cadeias em 0^*1^* que têm mais 0’s que 1’s, portanto aumentando o número de 0’s ainda dará uma cadeia em E . Nenhuma contradição ocorre. Precisamos tentar algo diferente.

O lema do bombeamento enuncia que $xy^iz \in E$ mesmo quando $i = 0$, portanto vamos considerar a cadeia $xy^0z = xz$. Remover a cadeia y diminui o número de 0’s em s . Lembre-se que s tem somente um 0 a mais que 1’s. Por conseguinte, xz não pode ter mais 0’s que 1’s, portanto ela não pode ser um membro de E . Por conseguinte obtemos uma contradição.

.....

Exercícios

- 1.1** Os diagramas a seguir são diagramas de estado de dois AFD’s, M_1 e M_2 . Responda às seguintes questões sobre essas máquinas.

1.4. LINGUAGENS NÃO-REGULARES 69

- a. Qual é o estado inicial de M_1 ?
- b. Qual é o conjunto de estados de aceitação de M_1 ?
- c. Qual é o estado de aceitação de M_2 ?
- d. Qual é o conjunto de estados de aceitação de M_2 ?
- e. Por qual seqüência de estados M_1 passa na entrada $aabb$?
- f. M_1 aceita a cadeia $aabb$?
- g. M_2 aceita a cadeia ε ?

1.2 Dê uma definição formal das máquinas M_1 e M_2 desenhadas no Exercício 1.1.

1.3 A descrição formal de um AFD M é $(\{q_1, q_2, q_3, q_4, q_5\}, \{u, d\}, \delta, q_3, \{q_3\})$, onde δ é dada pela tabela abaixo. Dê o diagrama de estados dessa máquina.

	u	d
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_2	q_4
q_4	q_3	q_5
q_5	q_4	q_5

1.4 Dê diagramas de estados de AFD's que reconhecem as linguagens a seguir. Em todos os casos o alfabeto é $\{0, 1\}$.

- a. $\{w \mid w \text{ começa com um } 1 \text{ e termina com um } 0\}$.
- b. $\{w \mid w \text{ contém pelo menos três } 1's\}$.
- c. $\{w \mid w \text{ contém a subcadeia } 0101, \text{ i.e., } w = x0101y \text{ para algum } x \text{ e algum } y\}$.
- d. $\{w \mid w \text{ tem comprimento pelo menos } 3 \text{ e seu terceiro símbolo é um } 0\}$.
- e. $\{w \mid w \text{ começa com um } 0 \text{ e tem comprimento ímpar, ou começa com um } 1 \text{ e tem comprimento par}\}$.
- f. $\{w \mid w \text{ não contém a subcadeia } 110\}$.
- g. $\{w \mid \text{o comprimento de } w \text{ é no máximo } 5\}$.
- h. $\{w \mid w \text{ é qualquer cadeia exceto } 11 \text{ e } 111\}$.
- i. $\{w \mid \text{toda posição ímpar de } w \text{ é um } 1\}$.
- j. $\{w \mid w \text{ contém pelo menos dois } 0's \text{ e no máximo um } 1\}$.
- k. $\{\varepsilon, 0\}$.
- l. $\{w \mid w \text{ contém um número par de } 0's, \text{ ou exatamente dois } 1's\}$.
- m. O conjunto vazio.
- n. Todas as cadeias exceto a cadeia vazia.

1.5 Dê AFN's com o número especificado de estados reconhecendo cada uma das seguintes linguagens:

- a. A linguagem $\{w \mid w \text{ termina com } 00 \text{ com três estados}\}$.
- b. A linguagem do Exercício 1.4c com cinco estados.
- c. A linguagem do Exercício 1.4l com seis estados.

- d.** A linguagem $\{0\}$ com dois estados.
 - e.** A linguagem $0^*1^*0^*0$ com três estados.
 - f.** A linguagem $\{\varepsilon\}$ com um estado.
 - g.** A linguagem 0^* com um estado.
- 1.6** Use a construção dada na prova do Teorema 1.22 para dar o diagrama de estados de AFN's reconhecendo a união
- a.** Exercícios 1.4a e 1.4b.
 - b.** Exercícios 1.4c e 1.4f.
- 1.7** Use a construção dada na prova do Teorema 1.23 para dar o diagrama de estados de AFN's reconhecendo a concatenação de linguagens descritas em
- a.** Exercícios 1.4g e 1.4i.
 - b.** Exercícios 1.4b e 1.4m.
- 1.8** Use a construção dada na prova do Teorema 1.24 para dar o diagrama de estados de AFN's reconhecendo a estrela de linguagens descritas em
- a.** Exercícios 1.4b.
 - b.** Exercícios 1.4j.
 - c.** Exercícios 1.4m.
- 1.9** Prove que todo AFN pode ser convertido para um equivalente que tem um único estado de aceitação.
- 1.10**
- a.** Mostre que, se M é um AFD que reconhece a linguagem B , trocando os estados de aceitação por estados de não-aceitação e vice-versa em M resulta em um novo AFD que reconhece o complemento de B . Conclua que a classe das linguagens regulares é fechada sob complemento.
 - b.** Mostre por meio de um exemplo que, se M é um AFN que reconhece a linguagem C , trocando os estados de aceitação por estados de não-aceitação e vice-versa em M não necessariamente resulta num novo AFN que reconhece o complemento de C . A classe das linguagens reconhecidas por AFN's é fechada sob complemento? Explique sua resposta.
- 1.11** Dê um contra-exemplo para mostrar que a seguinte construção falha em provar o Teorema 1.24, o fecho da classe de linguagens regulares sob a operação estrela.⁸ Suponha que $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ reconhece A_1 . Construa $N = (Q, \Sigma, \delta, q_1, F)$ da seguinte maneira. N supostamente reconhece A_1^* .
- a.** Os estados de N são os estados de N_1 .
 - b.** O estado inicial de N é o mesmo que o estado inicial de N_1 .
 - c.** $F = \{q_1\} \cup F_1$.
- Os estados de aceitação F são os antigos estados de aceitação mais seu estado inicial.

⁸Em outras palavras, você deve apresentar um autômato finito, N_1 para o qual o autômato construído N não reconhece a estrela da linguagem de N_1 .

- d. Defina δ de modo que para qualquer $q \in Q$ e qualquer $a \in \Sigma_\varepsilon$

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \notin F_1 \text{ ou } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1 \text{ e } a = \varepsilon. \end{cases}$$

(Sugestão: Converta essa construção formal em uma figura, como na Figura 1.26.)

- 1.12** Use a construção da no Teorema 1.19 para converter os autômatos (a) e (b) da Figura 1.37 em autômatos finitos determinísticos equivalentes.

- 1.13** Dê expressões regulares gerando as linguagens do Exercício 1.4.

- 1.14** Use o procedimento descrito no Lema 1.29 para converter as seguintes expressões regulares em autômatos finitos não-determinísticos.

- a. $(0 \cup 1)^* 000 (0 \cup 1)^*$
- b. $((((00)^*(11)) \cup 01)^*$
- c. \emptyset^*

- 1.15** Para cada uma das linguagens abaixo, dê duas cadeias que são membros e duas cadeias que *não* são membros—um total de quatro cadeias para cada item. Assuma que o alfabeto é $\Sigma = \{a, b\}$ em todos os itens.

- a. $a^* b^*$.
- b. $a(ba)^* b$.
- c. $a^* \cup b^*$.
- d. $(aaa)^*$.
- e. $\Sigma^* a \Sigma^* b \Sigma^* a \Sigma^*$.
- f. $aba \cup bab$.
- g. $(\varepsilon \cup a)b$.
- h. $(a \cup ba \cup bb)\Sigma^*$.

- 1.16** Use o procedimento descrito no Lema 1.32 para converter os autômatos finitos da Figura 1.38 para expressões regulares.

- 1.17** Use o lema do bombeamento para mostrar que as seguintes linguagens não são regulares.

- a. $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$.
- b. $A_2 = \{www \mid w \in \{a, b\}^*\}$.
- c. $A_3 = \{a^{2^n} \mid n \geq 0\}$. (Aqui, a^{2^n} significa uma cadeia de 2^n a's.)

- 1.18** Descreva o erro na seguinte “prova” de que $0^* 1^*$ não é uma linguagem regular. (Um erro tem que existir porque $0^* 1^*$ é regular.) A prova é por contradição. Assuma que $0^* 1^*$ é regular. Seja p o comprimento de bombeamento para $0^* 1^*$ dado pelo lema do bombeamento. Escolha s como sendo a cadeia $0^p 1^p$, mas o Exemplo 1.38 mostra que s não pode ser bombeada. Por conseguinte você tem uma contradição. Portanto $0^* 1^*$ não é regular.

- 1.19** Um *transdutor de estado finito* (TEF) é um tipo de autômato finito determinístico cuja saída é uma cadeia e não somente *aceita* ou *rejeita*. Os diagramas de estado da Figura 1.39 são diagramas de estado dos transdutores de estado finito T_1 e T_2 .

Cada transição de um TEF é rotulada com dois símbolos, um designando o símbolo de entrada e o outro designando o símbolo de saída. Os dois símbolos são escritos com uma barra, /, separando-os. Em T_1 , a transição de q_1 para q_2 tem símbolo de entrada 2 e símbolo de saída 1. Algumas transições podem ter múltiplos pares entrada-saída, tais como a transição em T_1 de q_1 para si próprio. Quando um TEF computa sobre uma cadeia de entrada w , ela toma os símbolos de entrada $w_1 \dots w_n$ um por um e, começando no estado inicial, segue as transições emparelhando os rótulos de entrada com a seqüência de símbolos $w_1 \dots w_n = w$. Toda vez que ele passa por uma transição, ele dá como saída o símbolo de saída correspondente. Por exemplo, sobre a entrada 2212011, a máquina T_1 entra na seqüência de estados $q_1, q_2, q_2, q_2, q_1, q_1, q_1, q_1$ e produz a saída 1111000. Sobre a entrada abbb, T_2 dá como saída 1011. Dê a seqüência de estados visitados e a saída produzida em cada uma das seguintes partes.

- a. T_1 sobre a entrada 011.
- b. T_1 sobre a entrada 211.
- c. T_1 sobre a entrada 0202.
- d. T_2 sobre a entrada b.
- e. T_2 sobre a entrada bbab.
- f. T_2 sobre a entrada bbbbb.
- g. T_2 sobre a entrada ε .

- 1.20** Leia a definição informal do transdutor de estados finitos dada no Exercício 1.19. Dê uma definição formal desse modelo, seguindo o padrão na Definição 1.1 na página 35. Assuma que um TEF tem um alfabeto de entrada Σ e um alfabeto de saída Γ mas não tem um conjunto de estados de aceitação. Inclua uma definição formal da computação de um TEF. (Dica: Um TEF é uma 5-upla. Sua função de transição é da forma $\delta : Q \times \Sigma \rightarrow Q \times \Gamma$.)
- 1.21** Usando a solução que você deu no Exercício 1.20, dê uma definição formal das máquinas T_1 e T_2 ilustradas no Exercício 1.19.
- 1.22** Leia a definição informal do transdutor de estados finitos dada no Exercício 1.19. Dê o diagrama de estados de um TEF com o seguinte comportamento. Seus alfabetos de entrada e de saída são $\{0, 1\}$. Sua cadeia de saída é idêntica à cadeia de entrada nas posições pares mas invertida nas posições ímpares. Por exemplo, sobre a entrada 0000111 ele deveria dar como saída 1010010.

Problemas

- 1.23** Prove que as seguintes linguagens não são regulares.

- a. $\{0^n 1^m 0^n \mid m, n \geq 0\}$.
- b. O complemento de $\{0^n 1^n \mid n \geq 0\}$.

- c. $\{0^m 1^n \mid m \neq n\}$.
- d. $\{w \mid w \in \{0, 1\}^*\text{ não é uma palíndrome}\}$.⁹

1.24 Para qualquer cadeia $w = w_1 w_2 \cdots w_n$, o *reverso* de w , escrito w^R , é a cadeia w na ordem reversa, $w_n \cdots w_2 w_1$. Para qualquer linguagem A , faça $A^R = \{w^R \mid w \in A\}$. Mostre que se A é regular, A^R também o é.

1.25 Seja

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Σ_3 contém todas as colunas de tamanho 3 de 0's e 1's. Uma cadeia de símbolos em Σ_3 dá três linhas de 0's e 1's. Considere cada linha como sendo um número binário e suponha que

$$B = \{w \in \Sigma_3^* \mid \text{a última linha de } w \text{ é a soma das duas primeiras linhas}\}.$$

Por exemplo,

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \in B, \quad \text{mas} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \notin B.$$

Mostre que B é regular. (Dica: Trabalhar com B^R é mais fácil. Você pode assumir o resultado afirmado no Problema 1.24.)

1.26 Seja

$$\Sigma_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

Aqui, Σ_2 contém todas as colunas de 0's e 1's de altura dois. Uma cadeia de símbolos em Σ_2 dá duas linhas de 0's e 1's. Considere cada linha como sendo um número binário e faça

$$C = \{w \in \Sigma_2^* \mid \text{a última linha de } w \text{ é três vezes a primeira linha}\}.$$

Por exemplo, $\begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \in C$, mas $\begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \notin C$. Mostre que C é regular. Você pode assumir o resultado afirmado no Problema 1.24.

1.27 Seja Σ_2 o mesmo que no Problema 1.26. Considere cada linha como sendo um número binário e faça

$$D = \{w \in \Sigma_2^* \mid \text{a primeira linha de } w \text{ é um número maior que o número na última linha}\}.$$

Por exemplo, $\begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \in D$, mas $\begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \notin D$. Mostre que D é regular.

⁹Uma *palíndrome* é uma cadeia que se lê da mesma forma de-frente-para-trás e de-trás-para-frente.

- 1.28** Seja Σ_2 o mesmo que no Problema 1.26. Considere a primeira e a última linhas como sendo cadeias de 0's e 1's e faça

$$E = \{w \in \Sigma_2^* \mid \text{a última linha de } w \text{ é o reverso da primeira linha de } w\}.$$

Mostre que E não é regular.

- 1.29** Seja $B_n = \{\alpha^k \mid \text{onde } k \text{ é um múltiplo de } n\}$. Mostre que para cada $n \geq 1$, a linguagem B_n é regular.

- 1.30** Seja $C_n = \{x \mid x \text{ é um número binário que é um múltiplo de } n\}$. Mostre que para cada $n \geq 1$, a linguagem C_n é regular.

- 1.31** Considere um novo tipo de autômato finito chamado AFN-de-todos-os-caminhos. Um AFN-de-todos-os-caminhos M é uma 5-upla $(Q, \Sigma, \delta, q_0, F)$ que aceita $x \in \Sigma^*$ se toda possível computação de M sobre x termina em um estado de F . Note, em contrapartida, que um AFN comum aceita uma cadeia se *alguma* computação termina em um estado de aceitação. Prove que AFN's-de-todos-os-caminhos reconhecem a classe de linguagens regulares.

- 1.32** Digamos que uma cadeia x é um *prefixo* de uma cadeia y se uma cadeia z existe onde $xz = y$ e que x é um *prefixo próprio* de y se adicionalmente $x \neq y$. Em cada um dos itens abaixo definimos uma operação sobre uma linguagem A . Mostre que a classe das linguagens regulares é fechada sob aquela operação.

- a. $\text{NÃOPREFIXO}(A) = \{w \in A \mid \text{nenhum prefixo próprio de } w \text{ é um membro de } A\}$.
- b. $\text{NÃOESTENDE}(A) = \{w \in A \mid \text{não é o prefixo próprio de nenhuma cadeia em } A\}$.

- 1.33** Leia a definição informal do transdutor de estados finitos dada no Exercício 1.19. Prove que nenhum TEF pode dar como saída w^R para toda entrada w se os alfabetos de entrada e de saída são $\{0, 1\}$.

- 1.34** Sejam x e y cadeias e suponha que L seja uma linguagem qualquer. Dizemos que x e y são *distingüíveis por* L se alguma cadeia z existe através da qual exatamente uma das cadeias xz e yz é um membro de L ; caso contrário, para toda cadeia z , $xz \in L$ sempre que $yz \in L$ e dizemos que x e y são *indistingüíveis por* L . Se x e y são indistingüíveis por L escrevemos $x \equiv_L y$. Mostre que \equiv_L é uma relação de equivalência.

- *1.35** Leia o Problema 1.34. Seja L uma linguagem e suponha que X seja um conjunto de cadeias. Digamos que X é *distingüível dois-a-dois por* L se quaisquer duas cadeias distintas em X são distingüíveis por L . Defina o *índice de* L como sendo o número máximo de elementos em qualquer conjunto que é distingüível dois-a-dois por L . O índice de L pode ser finito ou infinito.

- a. Mostre que se L é reconhecida por um AFD com k estados, L tem índice no máximo k .
- b. Mostre que se o índice de L é um número finito k , ela é reconhecida por um AFD com k estados.
- c. Conclua que L é regular sse la tem índice finito. Além do mais, seu índice é o tamanho do menor AFD que a reconhece.

1.36 Seja $\Sigma = \{0, 1, +, =\}$ e

$$SOMA = \{x=y+z \mid x, y, z \text{ são números binários, e } x \text{ é a soma de } y \text{ e } z\}.$$

Mostre que $SOMA$ não é regular.

1.37 Mostre que a linguagem $F = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ e se } i = 1 \text{ então } j = k\}$ satisfaz as três condições do lema do bombeamento muito embora ela não seja regular. Explique por que esse fato não contradiz o lema do bombeamento.

1.38 O lema do bombeamento diz que toda linguagem regular tem um comprimento de bombeamento p , tal que toda cadeia na linguagem pode ser bombeada se ela tem comprimento p ou mais. Se p é um comprimento de bombeamento para a linguagem A , o mesmo acontece com qualquer comprimento $p' \geq p$. O **comprimento mínimo de bombeamento** para A é o menor p que é um comprimento de bombeamento para A . Por exemplo, se $A = 01^*$, o comprimento mínimo de bombeamento é 2. A razão é que, a cadeia $s = 0 \in A$ e comprimento 1 não pode ser bombeada, e qualquer cadeia em A de comprimento 2 ou mais contém um 1 e portanto pode ser bombeada dividindo-a de modo que $x = 0$, $y = 1$, e z é o restante. Para cada uma das linguagens abaixo, dê o comprimento mínimo de bombeamento e justifique sua resposta.

- a. 0001^* .
- b. 0^*1^* .
- c. $(01)^*$.
- d. 01 .
- e. ϵ .

1.39 A construção no Teorema 1.28 mostra que todo AFNG é equivalente a um AFNG com apenas dois estados. Podemos mostrar que um fenômeno oposto ocorre para AFD's. Prove que para todo $k > 1$ uma linguagem $A_k \subseteq \{0, 1\}^*$ existe que é reconhecida por um AFD com k estados mas não por um com somente $k - 1$ estados.

***1.40** Se A é um conjunto de números naturais e k é um número natural maior que 1, faça

$$B_k(A) = \{w \mid w \text{ é a representação na base } k \text{ de algum número em } A\}.$$

Aqui, não permitimos 0's à esquerda na representação de um número. Por exemplo, $B_2(\{3, 5\}) = \{11, 101\}$ e $B_3(\{3, 5\}) = \{10, 12\}$. Dê um exemplo de um conjunto A para o qual $B_2(A)$ é regular mas $B_3(A)$ não é regular. Prove que seu exemplo funciona.

1.41 Seja

$$D = \{w \mid w \text{ contém igual número de ocorrências das subcadeias } 01 \text{ e } 10\}.$$

Por conseguinte $101 \in D$ porque 101 contém uma única ocorrência de 01 e uma única ocorrência de 10, mas $1010 \notin D$ pois 1010 contém duas 10 e uma 01. Mostre que D é uma linguagem regular.

- *1.42** Se A é uma linguagem regular qualquer, suponha que $A_{\frac{1}{2}-}$ seja o conjunto de todas as primeiras metades de cadeias em A de modo que

$$A_{\frac{1}{2}-} = \{x \mid \text{para alguma } y, |x| = |y|, \text{ e } xy \in A\}.$$

Mostre que, se A é regular, então $A_{\frac{1}{2}-}$ também o é.

- *1.43** Se A é uma linguagem regular qualquer, suponha que $A_{\frac{1}{3}-\frac{1}{3}}$ seja o conjunto de todas as cadeias em A com suas terças partes do meio removidas de modo que

$$A_{\frac{1}{3}-\frac{1}{3}} = \{xz \mid \text{para alguma } y, |x| = |y| = |z|, \text{ e } xyz \in A\}.$$

Mostre que, se A é regular, então $A_{\frac{1}{3}-\frac{1}{3}}$ também o é.

- *1.44** Dê uma família de linguagens E_n , onde cada E_n pode ser reconhecida por um AFN de n estados mas requer pelo menos c^n estados num AFD para alguma constante $c > 1$. Prove que suas linguagens têm essa propriedade.

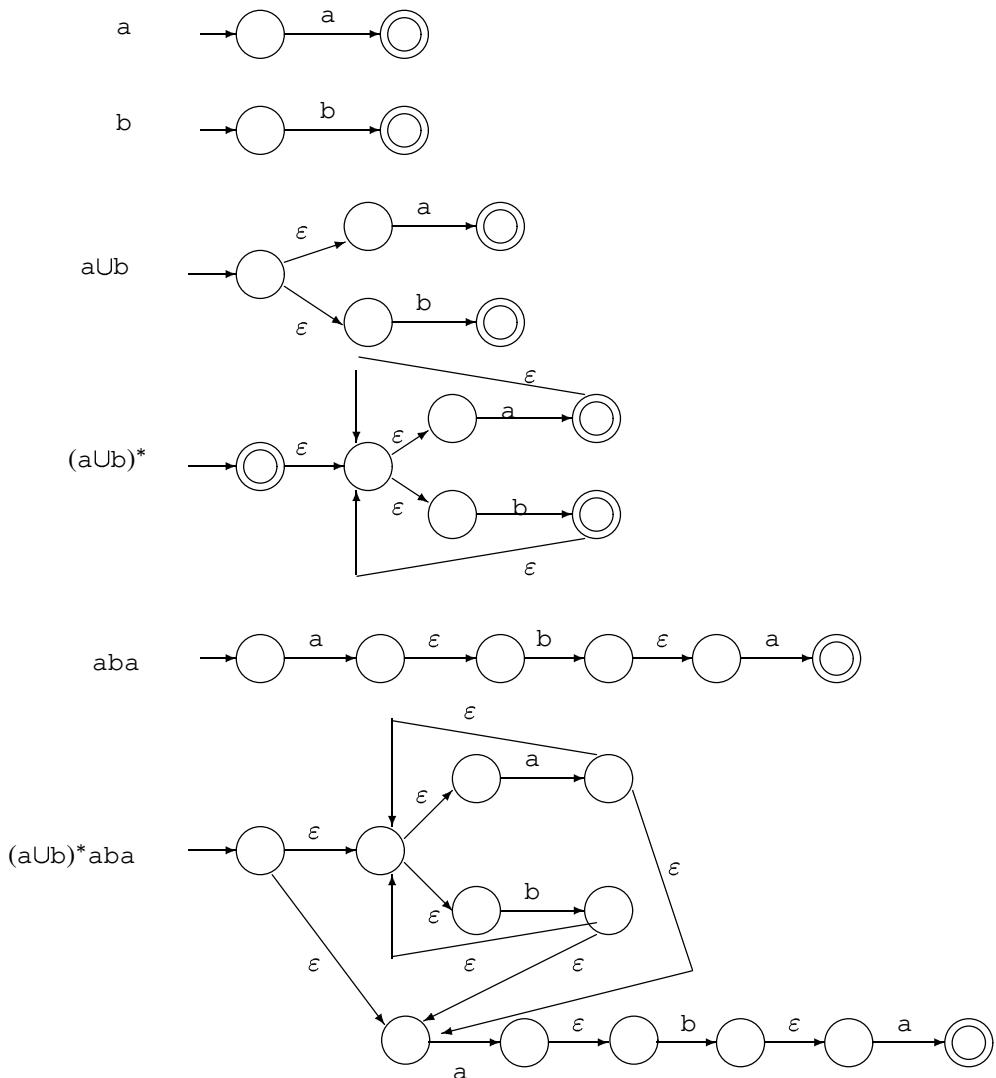


Figura 1.28: Construindo um AFN a partir da expressão regular $(aUb)^* aba$

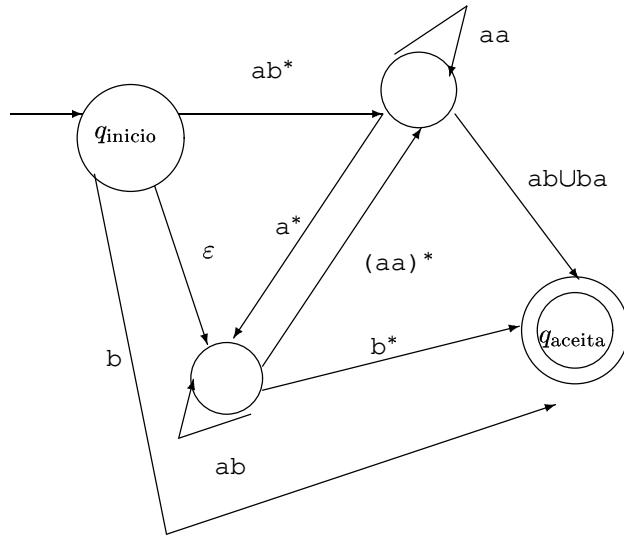


Figura 1.29: Um autômato finito não-determinístico generalizado

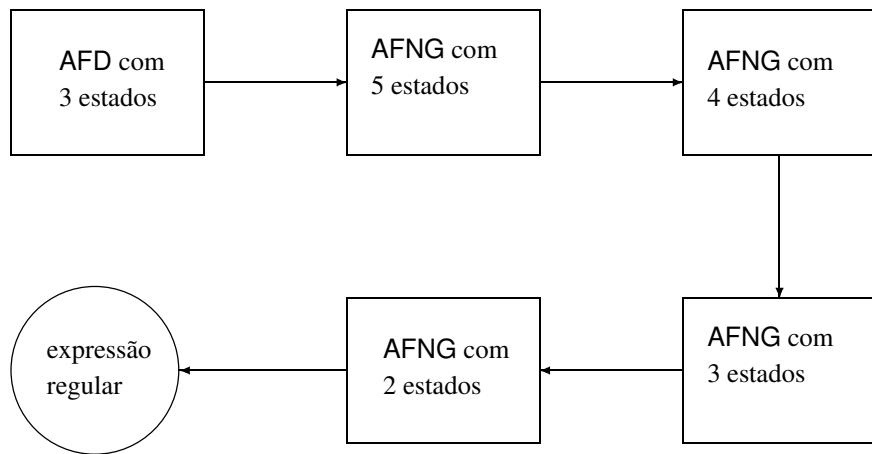


Figura 1.30: Estágios típicos na conversão de um AFD para uma expressão regular

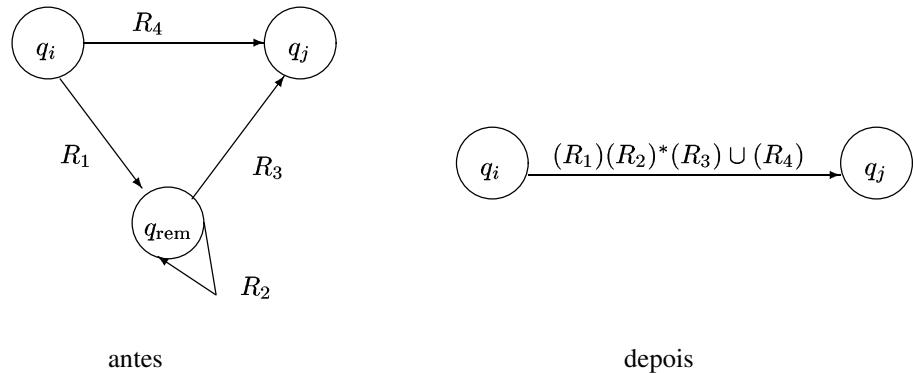


Figura 1.31: Construindo um AFNG equivalente com um estado a menos

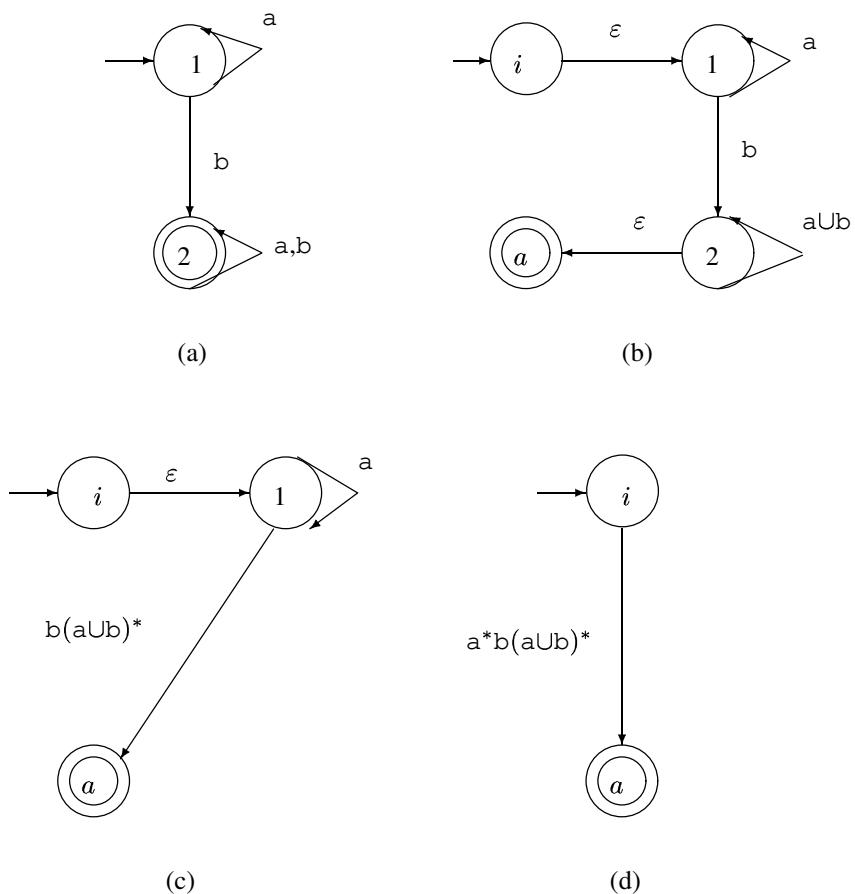


Figura 1.32: Convertendo um AFD de dois-estados em uma expressão regular equivalente

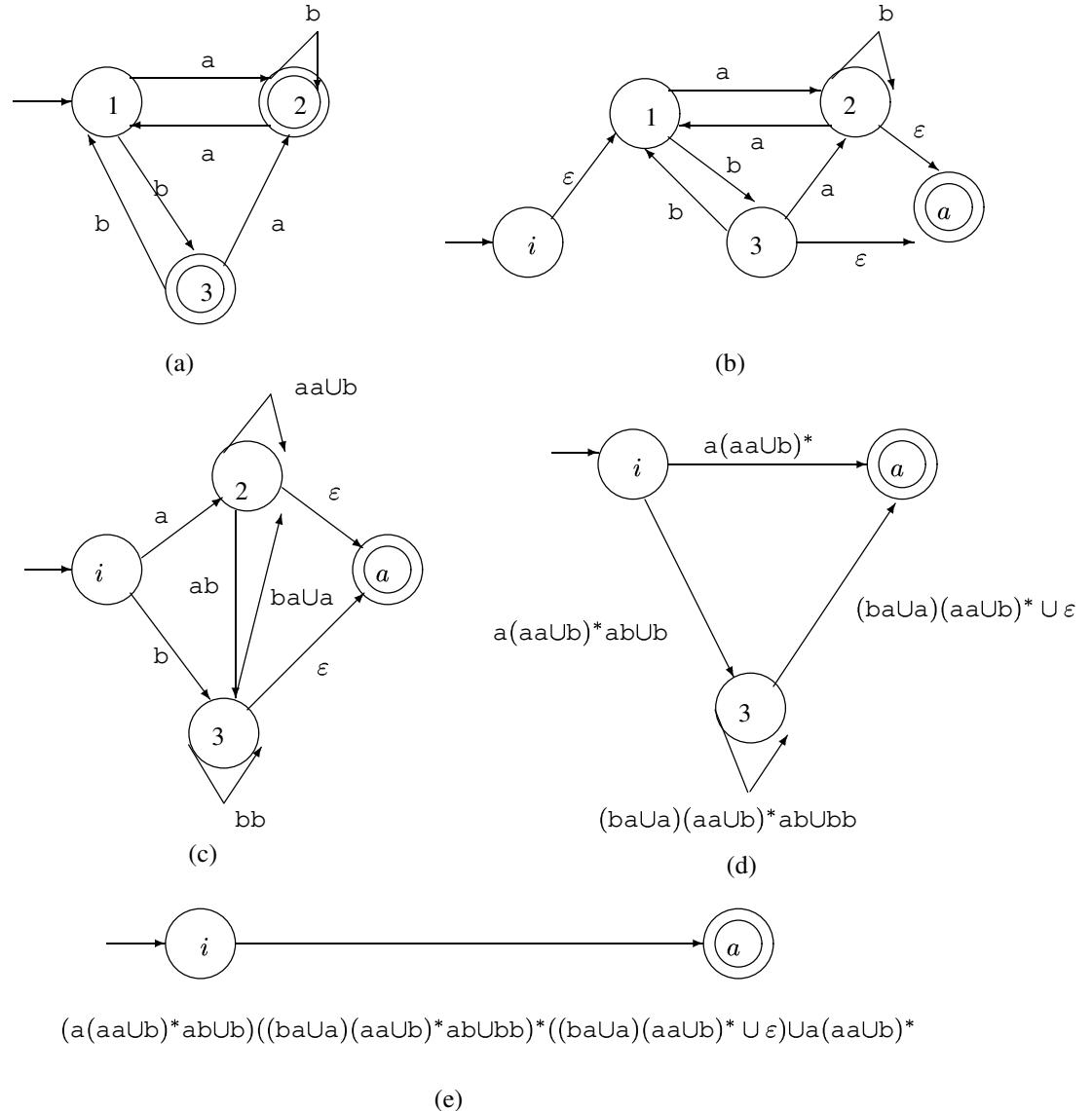


Figura 1.33: Convertendo um AFD de três-estados em uma expressão regular equivalente

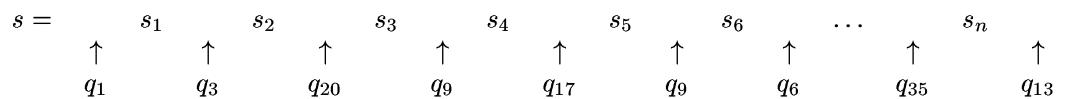


Figura 1.34: Exemplo mostrando o estado q_9 se repetindo quando M lê s

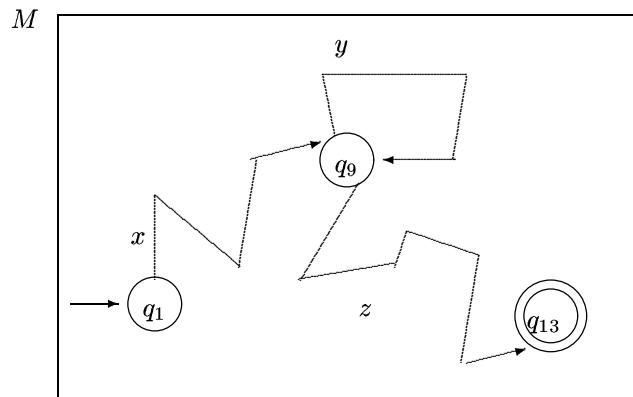


Figura 1.35: Exemplo mostrando como as cadeias x , y , e z afetam M

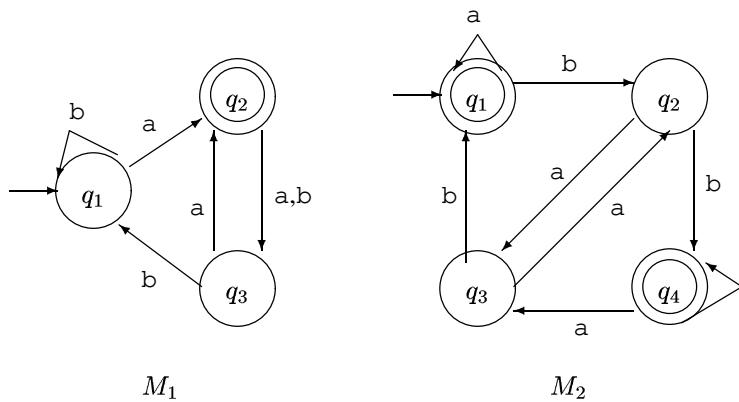


Figura 1.36:

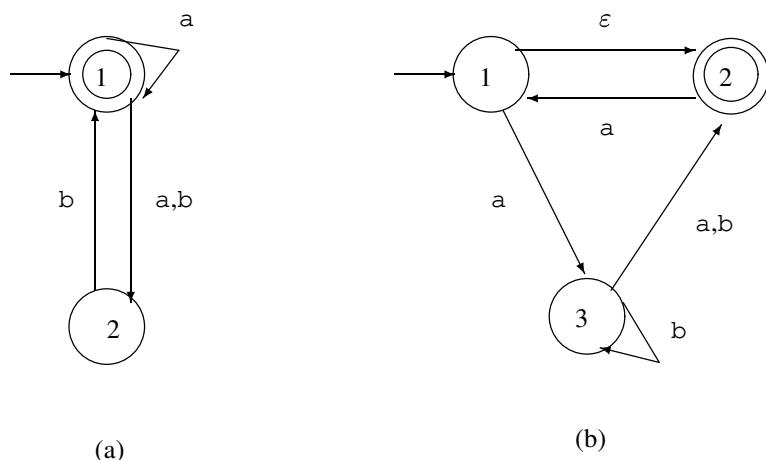
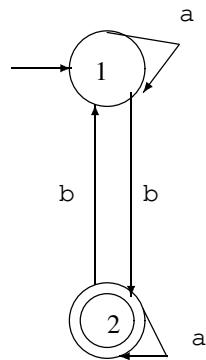
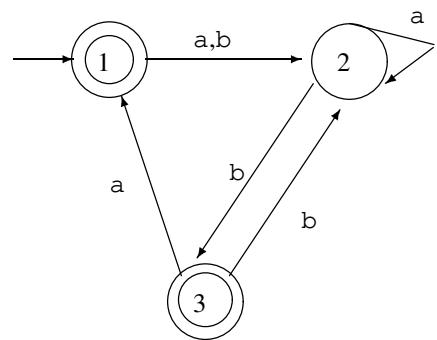


Figura 1.37:



(a)



(b)

Figura 1.38:

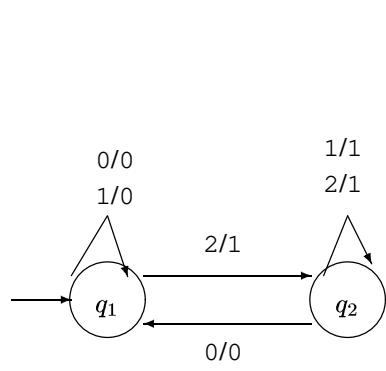
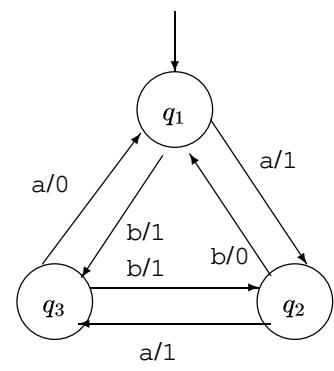
 T_1  T_2

Figura 1.39:

Capítulo 2

Linguagens Livres-do-Contexto

No Capítulo 1 introduzimos dois métodos diferentes, embora equivalentes, de descrever linguagens: *autômatos finitos* e *expressões regulares*. Mostramos que muitas linguagens podem ser descritas dessa maneira mas que algumas linguagens simples, tais como $\{0^n 1^n \mid n \geq 0\}$, não podem.

Neste capítulo introduzimos *gramáticas livres-do-contexto*, um método mais poderoso de descrever linguagens. Tais gramáticas podem descrever certas características que têm uma estrutura recursiva o que as torna úteis em uma variedade de aplicações.

Gramáticas livres-do-contexto foram primeiramente usadas no estudo de linguagens humanas. Uma maneira de entender o relacionamento de termos como *nome*, *verbo*, e *preposição* e suas respectivas frases leva a uma recursão natural porque frases nominais podem aparecer dentro de frases verbais e vice-versa. Gramáticas livres-do-contexto podem capturar aspectos importantes desses relacionamentos.

Uma aplicação importante de gramáticas livres-de-contexto ocorre na especificação e compilação de linguagens de programação. Uma gramática para uma linguagem de programação frequentemente aparece como uma referência para pessoas que estão tentando aprender a sintaxe da linguagem. Projetistas de compiladores e interpretadores para linguagens de programação frequentemente começam obtendo uma gramática para a linguagem. A maioria dos compiladores e interpretadores contêm uma componente chamada de *analisador* que extrai o significado de um programa antes de gerar o código compilado ou realizar a execução interpretada. Um número de metodologias facilitam a construção de um analisador uma vez que uma gramática livre-do-contexto está disponível. Algumas ferramentas até geram automaticamente o analisador a partir da gramática.

A coleção de linguagens associadas a gramáticas livres-do-contexto são chamadas as *linguagens livres-do-contexto*. Elas incluem todas as linguagens regulares e muitas linguagens adicionais. Neste capítulo, damos uma definição formal de gramáticas livres-do-contexto e estudamos as propriedades de linguagens livres-do-contexto. Também introduzimos *autômatos a pilha*, uma classe de máquinas que reconhece as linguagens livres-do-contexto. Autômatos a pilha são úteis porque eles nos permitem aumentar nosso conhecimento sobre o poder de gramáticas livres-do-contexto.

2.1 Gramáticas livres-do-contexro.....

O que segue é um exemplo de uma gramática livre-do-contexro, que chamaremos G_1 .

$$\begin{array}{l} A \rightarrow 0A1 \\ A \rightarrow B \\ B \rightarrow \# \end{array}$$

Uma gramática consiste de uma coleção de **regras de substituição**, também chamadas *produções*. Cada regra aparece como uma linha na gramática e compreende um símbolo e uma cadeia, separados por uma seta. O símbolo é chamado uma **variável**. A cadeia consiste de variáveis e outros símbolos chamados **terminais**. Os símbolos variáveis são frequentemente representados por letras maiúsculas. Os terminais são análogos ao alfabeto de entrada e frequentemente são representados por letras minúsculas, números, ou símbolos especiais. Uma variável é designada a **variável inicial**. Ela usualmente ocorre no lado esquerdo de uma das primeiras regras da gramática. Por exemplo, a gramática G_1 contém três regras. As variáveis de G_1 são A e B , onde A é a variável inicial. Seus terminais são $0, 1,$ e $\#$.

Você usa uma gramática para descrever uma linguagem gerando cada cadeia daquela linguagem da seguinte maneira:

1. Escreva a variável inicial. Ela é a variável no lado esquerdo da primeira regra, a menos que seja especificado o contrário.
2. Encontre uma variável que está escrita e uma regra que começa com aquela variável. Substitua a variável escrita pelo lado direito daquela regra.
3. Repita o passo 2 até que nenhuma variável permaneça.

Por exemplo, a gramática G_1 gera a cadeia $000\#\underline{1}\underline{1}\underline{1}$. A seqüência de substituições para obter uma cadeia é chamada uma **derivação**. Uma derivação da cadeia $000\#\underline{1}\underline{1}\underline{1}$ na gramática G_1 é

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Você pode também representar a mesma informação de uma maneira mais pictórica usando uma **árvore sintática**. Um exemplo de uma árvore sintática aparece na Figura 2.1.

Figura 2.1: Árvore sintática para $000\#\underline{1}\underline{1}\underline{1}$ na gramática G_1

Todas as cadeias geradas dessa maneira constituem a **linguagem da gramática**. Escrevemos $L(G_1)$ para a linguagem da gramática G_1 . Alguma experimentação com a gramática G_1 nos mostra que $L(G_1)$ é $\{0^n\#1^n \mid n \geq 0\}$. Qualquer linguagem que pode ser gerada por alguma gramática livre-do-contexro é chamada uma **linguagem livre-do-contexro** (LLC). Por conveniência ao apresentar uma gramática livre-do-contexro, abreviamos várias regras com a mesma variável do lado esquerdo, tais como $A \rightarrow 0A1$ e $A \rightarrow B$, em uma única linha $A \rightarrow 0A1 \mid B$, usando o símbolo “|” como um “ou.”

O que segue é um segundo exemplo de uma gramática livre-do-contexro chamada G_2 , que descreve um fragmento da língua inglesa.

$\langle \text{SENTENCE} \rangle$	\rightarrow	$\langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle$
$\langle \text{NOUN-PHRASE} \rangle$	\rightarrow	$\langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle$
$\langle \text{VERB-PHRASE} \rangle$	\rightarrow	$\langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle$
$\langle \text{PREP-PHRASE} \rangle$	\rightarrow	$\langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle$
$\langle \text{CMPLX-NOUN} \rangle$	\rightarrow	$\langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle$
$\langle \text{CMPLX-VERB} \rangle$	\rightarrow	$\langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle$
$\langle \text{ARTICLE} \rangle$	\rightarrow	a the
$\langle \text{NOUN} \rangle$	\rightarrow	boy girl flower
$\langle \text{VERB} \rangle$	\rightarrow	touches likes sees
$\langle \text{PREP} \rangle$	\rightarrow	with

A gramática G_2 tem dez variáveis (os termos gramaticais em maiúsculas escritos entre colchetes); 27 terminais (o alfabeto inglês padrão mais um caracter de espaço); e dezoito regras. Cadeias em $L(G_2)$ incluem os três exemplos seguintes.

a boy sees
 the boy sees a flower
 a girl with a flower likes the boy

Cada uma dessas cadeias tem uma derivação na gramática G_2 . O que segue é uma derivação da primeira cadeia nessa lista.

$\langle \text{SENTENCE} \rangle$	\Rightarrow	$\langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle$
	\Rightarrow	$\langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle$
	\Rightarrow	$\langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle$
	\Rightarrow	a $\langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle$
	\Rightarrow	a boy $\langle \text{VERB-PHRASE} \rangle$
	\Rightarrow	a boy $\langle \text{CMPLX-VERB} \rangle$
	\Rightarrow	a boy $\langle \text{VERB} \rangle$
	\Rightarrow	a boy sees

Definição formal de uma gramática livre-do-contex

Vamos formalizar nossa noção de uma gramática livre-do-contex (LLC).

Definição 2.1

Uma **gramática livre-do-contex** é uma 4-upla (V, Σ, R, S) , onde

1. V é um conjunto finito chamado as **variáveis**,
2. Σ é um conjunto finito, disjunto de V , chamado os **terminais**,
3. R é um conjunto finito de **regras**, com cada regra sendo uma variável e uma cadeia de variáveis e terminais, e
4. $S \in V$ é o símbolo inicial.

Se u, v , e w são cadeias de variáveis e terminais, e $A \rightarrow w$ é uma regra da gramática, dizemos que uAv **produz** uwv , escrito $A \Rightarrow uwv$. Escreva $u \xrightarrow{*} v$ se $u = v$ ou se uma seqüência u_1, u_2, \dots, u_k existe para $k \geq 0$ e

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v.$$

A **linguagem da gramática** é $\{w \in \Sigma^* \mid S \xrightarrow{*} w\}$.

Na gramática G_1 , $V = \{A, B\}$, $\Sigma = \{0, 1, \#\}$, $S = A$, e R é a coleção de três regras que aparecem na página ???. Na gramática G_2 ,

$$\begin{aligned} V = & \{\langle \text{SENTENCE} \rangle, \langle \text{NOUN-PHRASE} \rangle, \langle \text{VERB-PHRASE} \rangle, \\ & \langle \text{PREP-PHRASE} \rangle, \langle \text{CMPLX-NOUN} \rangle, \langle \text{CMPLX-VERB} \rangle, \\ & \langle \text{ARTICLE} \rangle, \langle \text{NOUN} \rangle, \langle \text{VERB} \rangle, \langle \text{PREP} \rangle\}. \end{aligned}$$

e $\Sigma = \{a, b, c, \dots, z, “ ”\}$. O símbolo “ ” é o símbolo branco, colocado invisivelmente entre cada palavra (a, boy, etc.), de forma que palavras não entram umas pelas outras.

Frequentemente especificamos uma gramática escrevendo apenas suas regras. Podemos identificar as variáveis como os símbolos que aparecem no lado esquerdo das regras e os terminais como os símbolos remanescentes. Por convenção, a variável inicial é a variável no lado esquerdo da primeira regra.

Exemplo 2.2

Considere a gramática $G_2 = (\{S\}, \{a, b\}, R, S)$. O conjunto de regras, R , é

$$S \rightarrow aSb \mid SS \mid \epsilon$$

Essa gramática gera cadeias tais como abab, aaabbb, aabbabb. Você pode ver mais facilmente o que essa linguagem é se você pensar em a como sendo um abre-parênteses “(” e b como um fecha-parênteses “)”. Vista dessa maneira, $L(G_3)$ é a linguagem de todas as cadeias de parênteses apropriadamente aninhados.

Exemplo 2.3

Considere a gramática $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

V é $\{\langle \text{EXPR} \rangle, \langle \text{TERMO} \rangle, \langle \text{FATOR} \rangle\}$ e Σ é $\{a, +, \times, (,)\}$. As regras são

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle + \langle \text{TERMO} \rangle \mid \langle \text{TERMO} \rangle \\ \langle \text{TERMO} \rangle &\rightarrow \langle \text{TERMO} \rangle \times \langle \text{FATOR} \rangle \mid \langle \text{FATOR} \rangle \\ \langle \text{FATOR} \rangle &\rightarrow (\langle \text{EXPR} \rangle) \mid a \end{aligned}$$

As duas cadeias $a+a \times a$ e $(a+a) \times a$ podem ser geradas com a gramática G_4 . As árvores sintáticas são mostradas na figura a seguir.

Figura 2.2: Árvores sintáticas para as cadeias $a+a \times a$ e $(a+a) \times a$

Um compilador traduz código escrito numa linguagem de programação para uma outra forma, usualmente uma forma mais adequada para execução. Para fazer isso o compilador extrai o significado do código para ser compilado em um processo chamado *parsing*. Uma representação desse significado é a árvore sintática para o código, no contexto da gramática livre-do-contexto para a linguagem de programação. Discutimos um algoritmo que analisa linguagens livres-do-contexto mais adiante no Teorema 7.14 e no Problema 7.38.

A gramática G_4 descreve um fragmento de uma linguagem de programação que lida com expressões aritméticas. Observe como as árvores sintáticas da Figura 2.2 “agrupam” as operações. A árvore para $a+a \times a$ agrupa o operador \times e seus operandos (os dois segundos a’s) juntos como um operando do operador $+$. Na árvore para

$(a+a) \times a$, o agrupamento é revertido. Esses agrupamentos estão de acordo com a precedência padrão da multiplicação antes da adição e o uso de parênteses para sobrepor a precedência padrão. A gramática G_4 é projetada para capturar essas relações de precedência.

Projetando gramáticas livres-do-contexro

Como no projeto de autômatos finitos, discutido na página 41 da Seção 1.1, o projeto de gramáticas livres-do-contexro requer criatividade. De fato, gramáticas livres-do-contexro são até mais complicadas de construir que autômatos finitos porque estamos mais acostumados a programar uma máquina para tarefas específicas do que a descrever linguagens com gramáticas. As seguintes técnicas são úteis, em separado ou em combinação, quando você se depara com o problema de construir uma GLC.

Primeiro, muitas GLC's são a união de GLC's mais simples. Se você tem que construir uma GLC para uma LLC que você pode partir em partes mais simples, faça isso e aí construa gramáticas separadas para cada parte. Essas gramáticas separadas podem ser facilmente combinadas em uma gramática para a linguagem original juntando todas as regras e variáveis iniciais das gramáticas separadas. Resolver vários problemas mais simples é frequentemente mais fácil que resolver um problema complicado.

Por exemplo, para obter uma gramática para a linguagem $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$, primeiro construa a gramática

$$S_1 \rightarrow 0S_11 \mid \varepsilon$$

para a linguagem $\{0^n 1^n \mid n \geq 0\}$ e a gramática

$$S_2 \rightarrow 1S_20 \mid \varepsilon$$

para a linguagem $\{1^n 0^n \mid n \geq 0\}$ e aí adicione as regras $S \rightarrow S_1 \mid S_2$ para dar a gramática

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_11 \mid \varepsilon \\ S_2 &\rightarrow 1S_20 \mid \varepsilon \end{aligned}$$

Segundo, construir uma GLC para uma linguagem que acontece de ser regular é fácil se você puder primeiro construir um AFD para aquela linguagem. Você pode converter um AFD em uma GLC equivalente da seguinte forma. Crie uma variável R_i para cada estado q_i do AFD. Adicione a regra $R_i \rightarrow aR_j$ à GLC se $\delta(q_i, a) = q_j$ é uma transição do AFD. Adicione a regra $R_i \rightarrow \varepsilon$ se q_i é um estado de aceitação do AFD. Crie a variável inicial R_0 da gramática, onde q_0 é o estado inicial da máquina. Verifique você mesmo que a GLC resultante gera a mesma linguagem que o AFD reconhece.

Terceiro, certas linguagens livres-do-contexro contêm cadeias com duas subcadeias que são “ligadas” no sentido de que uma máquina para uma tal linguagem necessitaria de memorizar uma quantidade ilimitada de informação sobre uma das subcadeias para verificar que ela corresponde propriamente à outra subcadeia. Essa situação ocorre na linguagem $\{0^n 1^n \mid n \geq 0\}$ porque uma máquina precisaria memorizar o número de 0's de modo a verificar que ele é igual ao número de 1's. Você pode construir uma GLC para lidar com essa situação usando uma regra da forma $R \rightarrow uRv$, que gera cadeias nas quais a parte contendo as u 's corresponde à parte contendo as v 's.

Finalmente, em linguagens mais complexas, as cadeias podem conter certas estruturas que aparecem recursivamente como parte de outras (ou as mesmas) estruturas.

Essa situação ocorre na gramática que gera expressões aritméticas no Exemplo 2.3. Em qualquer hora que o símbolo a aparece, um expressão parentizada inteira pode aparecer recursivamente ao invés do a . Para obter esse efeito, coloque a símbolo de variável gerando a estrutura no local das regras correspondendo a onde aquela estrutura pode recursivamente aparecer.

Ambigüidade

Às vezes uma gramática pode gerar a mesma cadeia de diversas maneiras diferentes. Tal cadeia terá árvores sintáticas diferentes e por conseguinte vários significados diferentes. Esse resultado pode ser indesejável para certas aplicações, tais como linguagens de programação, onde um dado programa deve ter uma única interpretação.

Se uma gramática gera a mesma cadeia de várias maneiras diferentes, dizemos que a cadeia é derivada *ambiguamente* naquela gramática. Se uma gramática gera alguma cadeia ambiguamente dizemos que a gramática é *ambígua*.

Por exemplo, vamos considerar a gramática G_5 :

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid a$$

Essa gramática gera a cadeia $a+a \times a$ ambiguamente. A seguinte figura mostra as duas árvores sintáticas.

Figura 2.3: As duas árvores sintáticas para a cadeia $a+a \times a$

Essa gramática não captura as relações de precedência usuais e portanto pode agrupar o $+$ antes do \times ou vice-versa. Diferentemente, a gramática G_4 gera exatamente a mesma linguagem, mas toda cadeia gerada tem uma única árvore sintática. Portanto G_4 é não-ambígua, enquanto que G_5 é ambígua.

A gramática G_2 da página 93 é um outro exemplo de uma gramática ambígua. A sentença *the girl touches the boy with the flower* tem duas derivações diferentes. No Exercício 2.8 pede-se para dar as duas árvores sintáticas e observar sua correspondência com as duas maneiras diferentes de ler aquela sentença.

Agora formalizamos a noção de ambigüidade. Quando dizemos que uma gramática gera uma cadeia ambiguamente, queremos dizer que a cadeia tem duas árvores sintáticas diferentes, e não duas derivações diferentes. Duas derivações podem diferir meramente na ordem em que elas substituem variáveis e mesmo assim não diferir na sua estrutura global. Para nos concentrarmos sobre a estrutura definimos um tipo de derivação que substitui variáveis numa ordem fixada. Uma derivação de uma cadeia w em uma gramática G é uma **derivação à esquerda** se a cada passo a variável mais à esquerda remanescente é aquela a ser substituída. A derivação na página 94 é uma derivação à esquerda.

Definição 2.4

Uma cadeia w é derivada **ambiguamente** na gramática livre-do-contexro G se ela tem duas ou mais derivações à esquerda. A gramática G é **ambígua** se ela gera alguma cadeia ambiguamente.

2.1. GRAMÁTICAS LIVRES-DO-CONTEXTO 89

Às vezes quando temos uma gramática ambígua podemos encontrar uma gramática não-ambígua que gera a mesma linguagem. Algumas linguagens livres-do-contesto, entretanto, somente podem ser geradas por gramáticas ambíguas. Tais linguagens são chamadas *inherentemente ambíguas*. O Problema 2.24 pede para você provar que a linguagem $\{0^i 1^j 2^k \mid i = j \text{ ou } j = k\}$ é inherentemente ambígua.

Forma normal de Chomsky

Quando se trabalha com gramáticas livres-do-contesto, é frequentemente conveniente se tê-las em forma simplificada. Uma das formas mais simples e mais úteis é chamada a forma normal de Chomsky. Vamos achar a forma normal de Chomsky útil quando estivermos dando algoritmos para trabalhar com gramáticas livres-do-contesto nos Capítulos 4 e 7.

Definição 2.5

Uma gramática livre-do-contesto está na **forma normal de Chomsky** se toda regra está na forma

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

onde a é qualquer terminal e A, B , e C são quaisquer variáveis—exceto que B e C não podem ser a variável inicial. Adicionalmente permitimos a regra $S \rightarrow \varepsilon$, onde S é a variável inicial.

Teorema 2.6

Qualquer linguagem livre-do-contesto é gerada por uma gramática na forma normal de Chomsky.

Idéia da prova. Podemos converter qualquer gramática G na forma normal de Chomsky. A conversão tem diversos estágios nos quais as regras que violam as condições são substituídas por regras equivalentes que são satisfatórias. Primeiro, adicionamos um novo símbolo inicial. Então, eliminamos todas as **regras ε** da forma $A \rightarrow \varepsilon$. Também eliminamos todas as **regras unitárias** da forma $A \rightarrow B$. Em ambos os casos a gramática é então consertada para garantir que ela ainda gera a mesma linguagem. Finalmente, convertemos as regras remanescentes na forma apropriada.

Prova. Primeiro, adicionamos um novo símbolo inicial S_0 e a regra $S_0 \rightarrow S$, onde S era o símbolo inicial original. Essa mudança garante que o símbolo inicial não ocorre no lado direito de uma regra.

Segundo, cuidamos de todas as regras ε . Removemos uma ε -regra $A \rightarrow \varepsilon$, onde A não é o símbolo inicial. Então para cada ocorrência de um A no lado direito de uma regra, adicionamos uma nova regra com aquela ocorrência apagada. Em outras palavras, se $R \rightarrow uAv$ é uma regra na qual u e v são cadeias de variáveis e terminais, adicionamos a regra $R \rightarrow uv$. Fazemos isso para cada *ocorrência* de um A , de modo que a regra $R \rightarrow uAvAw$ nos leve a adicionar $R \rightarrow uvAw$, $R \rightarrow uAvw$, e $R \rightarrow uvw$. Se tivermos a regra $R \rightarrow A$, adicionamos $R \rightarrow \varepsilon$ a menos que tivéssemos removido previamente a regra $R \rightarrow \varepsilon$. Repetimos esses passos até que eliminemos todas as regras ε que não envolvem a variável inicial.

Terceiro, manuseamos as regras unitárias. Removemos uma regra unitária $A \rightarrow B$. Então, sempre que uma regra $B \rightarrow u$ aparece, adicionamos a regra $A \rightarrow u$ a menos

que essa fosse uma regra unitária previamente removida. Como antes, u é uma cadeia de variáveis e terminais. Repetimos esses passos até que eliminemos todas as regras unitárias.

Finalmente, convertemos todas as regras remanescentes para a forma apropriada. Substituimos cada regra $A \rightarrow u_1u_2 \dots u_k$ onde $k \geq 3$ e cada u_i é uma variável ou um símbolo terminal, com as regras $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2$, $A_2 \rightarrow u_3A_3$, ..., $A_{k-2} \rightarrow u_{k-1}u_k$. Os A_i 's são novas variáveis. Se $k \geq 2$, substituimos qualquer terminal u_i na(s) regra(s) precedente(s) por novas variáveis U_i e adicionamos a regra $U_i \rightarrow u_i$.

.....

Exemplo 2.7

Suponha que G_6 seja a GLC a seguir e converta-a para a forma normal de Chomsky usando o procedimento de conversão que acaba de ser dado. A seguinte série de gramáticas ilustra os passos na conversão. As regras sublinhadas acabaram de ser adicionadas. As regras mostradas em $\boxed{}$ acabaram de ser removidas.

1. A GLC G_6 original é mostrada à esquerda. O resultado de se aplicar o primeiro passo para acrescentar um novo símbolo original aparece à direita.

$$\begin{array}{ll} S_0 \rightarrow S & \\ S \rightarrow ASA | aB & S \rightarrow ASA | aB \\ A \rightarrow B | S & A \rightarrow B | S \\ B \rightarrow b | \varepsilon & B \rightarrow b | \varepsilon \end{array}$$

2. Remova as regras ε , mostradas à esquerda, e $A \rightarrow \varepsilon$, mostrada à direita.

$$\begin{array}{ll} S_0 \rightarrow S & S_0 \rightarrow S \\ S \rightarrow ASA | aB | a & S \rightarrow ASA | aB | a | \mathbf{SA} | \mathbf{AS} | S \\ A \rightarrow B | S | \varepsilon & A \rightarrow B | S | \boxed{\varepsilon} \\ B \rightarrow b | \boxed{\varepsilon} & B \rightarrow b \end{array}$$

- 3a. Remova as regras unitárias $S \rightarrow S$, mostradas à esquerda, e $S_0 \rightarrow S$, mostrada à direita.

$$\begin{array}{ll} S_0 \rightarrow S & S_0 \rightarrow \boxed{S} | ASA | aB | a | \mathbf{SA} | \mathbf{AS} \\ S \rightarrow ASA | aB | a | SA | AS | \boxed{S} & S \rightarrow ASA | aB | a | \mathbf{SA} | \mathbf{AS} \\ A \rightarrow B | S & A \rightarrow B | S \\ B \rightarrow b & B \rightarrow b \end{array}$$

- 3b. Remova as regras unitárias $A \rightarrow B$ e $A \rightarrow S$.

$$\begin{array}{ll} S_0 \rightarrow ASA | aB | a | SA | AS & S_0 \rightarrow ASA | aB | a | SA | AS \\ S \rightarrow ASA | aB | a | SA | AS & S \rightarrow ASA | aB | a | SA | AS \\ A \rightarrow B | S | b & A \rightarrow S | b | \mathbf{ASA} | \mathbf{aB} | a | \mathbf{SA} | \mathbf{AS} \\ B \rightarrow b & B \rightarrow b \end{array}$$

4. Converte as regras remanescentes na forma apropriada acrescentando variáveis e regras adicionais. A gramática final na forma normal de Chomsky é equivalente a G_6 e aparece a seguir. (Na verdade o procedimento dado no Teorema 2.6 produz diversas variáveis U_i juntamente com várias regras $U_i \rightarrow a$. Simplificamos a gramática resultante usando uma única variável U e uma única regra $U \rightarrow a$.)

$$\begin{aligned}
 S_0 &\rightarrow AA_1 | UB | a | SA | AS \\
 S &\rightarrow AA_1 | UB | a | SA | AS \\
 A &\rightarrow b | AA_1 | UB | a | SA | AS \\
 A_1 &\rightarrow SA \\
 U &\rightarrow a \\
 B &\rightarrow b
 \end{aligned}$$

2.2 Autômatos a pilha

Nesta seção introduzimos um novo tipo de modelo computacional chamado **autômato a pilha**. Esses autômatos são como autômatos finitos não-determinísticos mas têm um componente extra chamado uma **pilha**. A pilha provê memória adicional além do autômato finito disponível no controle. A pilha permite autômatos a pilha reconhecerem algumas linguagens não-regulares.

Autômatos a pilha são equivalentes em poder a gramáticas livres-do-contexto. Essa equivalência é útil porque ela nos dá duas opções para provar que uma linguagem é livre-do-contexto. Podemos ou dar uma gramática livre-do-contexto que a gera ou um autômato a pilha que a reconhece. Certas linguagens são mais facilmente descritas em termos de geradores, enquanto que outras são mais facilmente descritas em termos de reconhecedores.

A figura a seguir é uma representação esquemática de um autômato finito. O controle representa os estados e a função de transição, a fita contém a cadeia de entrada, e a seta representa a cabeça de entrada, apontando para o próximo símbolo de entrada a ser lido.

Figura 2.4: Esquemática de um autômato finito

Com a adição de um componente pilha obtemos uma representação esquemática de um autômato a pilha, como mostrado na figura a seguir.

Figura 2.5: Esquemática de um autômato a pilha

Um autômato a pilha (**AP**) pode escrever símbolos na pilha e lê-los de volta mais adiante. Escrever um símbolo “empilha” todos os outros símbolos sobre a pilha. A qualquer tempo o símbolo no topo da pilha pode ser lido e removido. Os símbolos remanescentes então sobem. O ato de escrever um símbolo na pilha é frequentemente referido como **empilhar** o símbolo, e o ato de remover um símbolo é referido como

desempilhá-lo. Note que todo acesso à pilha, tanto para ler quanto para escrever, pode ser feito somente no topo. Em outras palavras, uma pilha é um dispositivo de armazenamento “último que entra, primeiro que sai”. Se uma certa informação for escrita na pilha e informação adicional for escrita depois, a informação mais antiga torna-se inacessível até que a informação mais nova seja removida.

Os pratos num balcão de servir de uma cafeteria ilustram uma pilha. A pilha de pratos repousa sobre uma fonte de modo que quando um novo prato é colocado sobre a pilha, os pratos abaixo dele descem. A pilha em um autômato a pilha é como uma pilha de pratos, com cada prato tendo um símbolo escrito nele.

Uma pilha é valorosa porque ela pode armazenar uma quantidade ilimitada de informação. Lembre-se que um autômato finito é incapaz de reconhecer a linguagem $\{0^n 1^n \mid n \geq 0\}$ porque ele não pode armazenar números muito grandes de 0's que ele tenha visto. Por conseguinte a natureza ilimitada de uma pilha permite ao AP armazenar números de tamanho ilimitado. A seguinte descrição informal mostra como o autômato para essa linguagem funciona.

Leia símbolos da entrada. À medida que cada 0 é lido, empilhe-o na pilha. Assim que 1's são vistos, desempilhe um 0 da pilha para cada 1 lido. Se a leitura da entrada termina exatamente quanto a pilha fica vazia de 0's, aceite a entrada. Se a pilha fica vazia enquanto 1's permanecem ou se os 1's terminam enquanto a pilha ainda contém 0's ou se quaisquer 0's aparecem na entrada seguindo 1's, rejeite a entrada.

Como mencionado anteriormente, autômatos a pilha podem ser não-determinísticos. Essa característica é crucial porque, ao contrário da situação dos autômatos finitos, não-determinismo adiciona poder à capacidade que autômatos a pilha teriam se lhes permitissem somente ser determinísticos. Algumas linguagens, tais como $\{0^n 1^n \mid n \geq 0\}$, não requerem não-determinismo, mas outras sim. Damos uma linguagem que requer não-determinismo no Exemplo 2.11.

Definição formal de um autômato a pilha

A definição formal de um autômato a pilha é semelhante àquela de um autômato finito exceto pela pilha. A pilha é um dispositivo contendo símbolos tirados de algum alfabeto. A máquina pode usar alfabetos diferentes para sua entrada e sua pilha, portanto agora especificamos tanto o alfabeto de entrada Σ quanto o alfabeto da pilha Γ .

No coração de qualquer definição formal de um autômato está a função de transição, pois ela descreve seu comportamento. Lembre-se que $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. O domínio da função de transição é $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Por conseguinte o estado atual, o próximo símbolo de entrada a ser lido, e o símbolo no topo da pilha determinam o próximo movimento de um autômato a pilha. O símbolo pode ser ε causando a máquina a mover sem ler um símbolo da entrada, ou a mover sem ler um símbolo da pilha.

Para o codomínio da função de transição precisamos considerar o que permite ao autômato fazer quando ele está numa dada situação. Ele pode entrar em algum novo estado e possivelmente escrever um símbolo no topo da pilha. A função δ pode indicar essa ação retornando um membro de Q juntamente com um membro de Γ_ε , ou seja, um membro de $Q \times \Gamma_\varepsilon$. Devido ao fato de que permitimos não-determinismo nesse modelo, uma situação pode ter várias movimentos próximos legais. A função de transição incorpora não-determinismo da maneira usual, retornando um conjunto de membros de $Q \times \Gamma_\varepsilon$, ou seja, um membro de $\mathcal{P}(Q \times \Gamma_\varepsilon)$. Colocando tudo junto, nossa função de transição δ toma a forma $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.

Definição 2.8

Um **autômato a pilha** é uma 6-upla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, onde Q, Σ, Γ , e F são todos conjuntos finitos, e

1. Q é o conjunto de estados,
2. Σ é o alfabeto de entrada,
3. Γ é o alfabeto da pilha,
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ é a função de transição,
5. $q_0 \in Q$ é o estado inicial, e
6. $F \subseteq Q$ é o conjunto de estados de aceitação.

Um autômato a pilha $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computa da seguinte forma. Ele aceita a entrada w se w pode ser escrita como $w = w_1 w_2 \dots w_n$, onde cada $w_i \in \Sigma_\varepsilon$ e seqüências de estados $r_0, r_1, \dots, r_m \in Q$ e cadeias $s_0, s_1, \dots, s_m \in \Gamma^*$ existem que satisfazem as três condições abaixo. As cadeias s_i representam a seqüência de conteúdos da pilha que M tem no ramo de aceitação da computação.

1. $r_0 = q_0$ e $s_0 = \varepsilon$. Esta condição significa que M começa de forma apropriada, no estado inicial e com a pilha vazia.
2. Para $i = 0, \dots, m - 1$, temos $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, onde $s_i = at$ e $s_{i+1} = bt$ para algum $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$. Esta condição enuncia que M move apropriadamente conforme o estado, a pilha, e o próximo símbolo de entrada.
3. $r_m \in F$. Essa condição enuncia que um estado de aceitação ocorre no final da entrada.

Exemplos de autômatos a pilha**Exemplo 2.9**

O que segue é uma descrição formal do AP da página ?? que reconhece a linguagem $\{0^n 1^n \mid n \geq 0\}$. Suponha que M_1 seja $(Q, \Sigma, \Gamma, \delta, q_1, F)$ onde

$$Q = \{q_1, q_2, q_3, q_4\},$$

$$\Sigma = \{0, 1\},$$

$$\Gamma = \{0, \$\},$$

$$F = \{q_1, q_4\},$$

e δ seja dada pela seguinte tabela, na qual uma entrada em branco significa \emptyset .

Entrada:	0			1			ε		
Pilha:	0	\$	ε	0	\$	ε	0	\$	ε
q_1							$\{(q_2, \$)\}$		
q_2	$\{(q_2, 0)\}$			$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$					
q_4							$\{(q_4, \varepsilon)\}$		

Podemos também usar um diagrama de estados para descrever um AP, como mostrado nas Figuras 2.6, 2.7 e 2.8. Tais diagramas são semelhantes aos diagramas de estado usados para descrever autômatos finitos, modificados para mostrar como o AP usa sua pilha quando vai de um estado para outro. Escrevemos “ $a, b \rightarrow c$ ” para significar que quando a máquina está lendo um a da entrada ela pode substituir o símbolo b no topo da pilha por um c . Quaisquer dos a , b , e c pode ser ϵ . Se a é ϵ , a máquina pode fazer sua transição sem ler qualquer símbolo da entrada. Se b é ϵ , a máquina não escreve qualquer símbolo na pilha quando faz essa transição.

Figura 2.6: Diagrama de estados para o AP M_1 que reconhece $\{0^n 1^n \mid n \geq 0\}$

A definição formal de um AP não contém qualquer mecanismo explícito para permitir que o AP teste por uma pilha vazia. Esse AP é capaz de obter o mesmo efeito colocando inicialmente um símbolo especial $\$$ na pilha. Então se ele por acaso vê $\$$ novamente, ele sabe que a pilha efetivamente está vazia. Subsequentemente, quando nos referimos a testar por uma pilha vazia em uma descrição informal de um AP, implementamos o procedimento da mesma maneira.

Igualmente, AP's não podem testar explicitamente por ter atingido o final da cadeia de entrada. Esse AP é capaz de alcançar esse efeito porque o estado de aceitação toma efeito somente quando a máquina está no final da cadeia. Por conseguinte de agora por diante, assumimos que AP's podem testar pelo final da entrada, e sabemos que podemos implementá-lo da mesma maneira.

Exemplo 2.10

Este exemplo ilustra um autômato a pilha que reconhece a linguagem

$$\{a^i b^j c^k \mid i, j, k \geq 0 \text{ e } i = j \text{ ou } i = k\}.$$

Informalmente o AP para essa linguagem funciona primeiro lendo e empilhando os a 's. Quando os a 's terminam a máquina tem todos eles na pilha de modo que ela pode emparelhá-los ou com os b 's ou com os c 's. Essa manobra é um pouco complicada porque a máquina não sabe com antecipação se emparelha os a 's com os b 's ou com os c 's. Não-determinismo vem para ajudar aqui.

Usando seu não-determinismo, o AP pode adivinhar se emparelha os a 's com os b 's ou com os c 's, como mostrado na Figura 2.7. Pense na máquina como tendo dois ramos de seu não-determinismo, um para cada palpite possível. Se um deles emparelha, aquele ramo aceita e a máquina inteira aceita. Na verdade poderíamos mostrar, embora não o fazemos, que não-determinismo é *essencial* para reconhecer essa linguagem com um AP.

Figura 2.7: Diagrama de estados para o AP M_2 que reconhece $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ e } i = j \text{ ou } i = k\}$

Exemplo 2.11

Neste exemplo damos um AP M_3 que reconhece a linguagem $\{ww^R \mid w \in \{0,1\}^*\}$. Lembre-se que w^R significa w escrita de trás para frente. A descrição informal do AP segue.

Comece empilhando os símbolos que são lidos na pilha. A cada ponto não-deterministicamente estipule que o meio da cadeia foi atingido e então mude para desempilhar a pilha para cada símbolo lido, verificando para ver se eles são os mesmos. Se eles fossem sempre o mesmo símbolo e a pilha se esvaziasse ao mesmo tempo em que a entrada termina, aceite; caso contrário, rejeite.

A seguir está o diagrama dessa máquina.

Figura 2.8: Diagrama de estados para o AP M_3 que reconhece $\{ww^R \mid w \in \{0,1\}^*\}$

Equivalência com gramáticas livres-do-contex

Nesta seção mostramos que gramáticas livres-do-contex e autômatos a pilha são equivalentes em poder. Ambos são capazes de descrever a classe das linguagens livres-do-contex. Mostramos como converter qualquer gramática livre-do-contex em um autômato a pilha que reconhece a mesma linguagem e vice-versa. Relembrando que definimos uma linguagem livre-do-contex como sendo qualquer linguagem que pode ser descrita por uma gramática livre-do-contex, nosso objetivo é o seguinte teorema.

Teorema 2.12

Uma linguagem é livre-do-contex se e somente se algum autômato a pilha a reconhece.

Como de costume para teoremas “se e somente se”, temos duas direções para provar. Neste teorema, ambas as direções são interessantes. Primeiro, vamos fazer a parte mais fácil que é a ida.

Lema 2.13

Se uma linguagem é livre-do-contex, então algum autômato a pilha a reconhece.

Idéia da prova. Seja A uma LLC. Da definição sabemos que A tem uma GLC, G , que a gera. Mostramos como converter G em um AP equivalente, que chamamos P .

O AP P que agora descrevemos funcionará aceitando sua entrada w , se G gera aquela entrada, determinando se existe uma derivação para w . Lembre-se que uma derivação é simplesmente a seqüência de substituições feitas à medida que a gramática gera uma cadeia. Cada passo da derivação dá origem a uma **cadeia intermediária** de variáveis e terminais. Projetamos P para determinar alguma série de substituições usando as regras G podem levar da variável inicial para w .

Uma das dificuldades em se testar se existe uma derivação para w está em descobrir quais substituições fazer. O não-determinismo do AP o permite estipular a seqüência

de substituições corretas. A cada passo da derivação uma das regras para uma dada variável é selecionada não-deterministicamente e usada para substituir aquela variável.

O AP P começa escrevendo a variável inicial na sua pilha. Ela passa por uma série de cadeias intermediárias, fazendo uma substituição após outra. Em algum ponto ele pode chegar a uma cadeia que contém somente símbolos terminais, o que quer dizer que ele derivou uma cadeia usando a gramática. Então P aceita se essa cadeia for idêntica à cadeia que ela recebeu como entrada.

Implementar essa estratégia em um AP requer uma idéia adicional. Precisamos ver como o AP armazena as cadeias intermediárias à medida que ela vai de uma para outra. Simplesmente usar a pilha para armazenar cada cadeia intermediária é tentador. Entretanto, isso não chega a funcionar porque o AP precisa encontrar as variáveis na cadeia intermediária e fazer substituições. O AP pode acessar somente o símbolo no topo da pilha e esse pode ser um símbolo terminal ao invés de uma variável. A forma de contornar esse problema é manter somente *parte* da cadeia intermediária na pilha: os símbolos começando com a primeira variável na cadeia intermediária. Quaisquer símbolos terminais aparecendo antes da primeira variável são emparelhados imediatamente com símbolos na cadeia de entrada. A figura abaixo mostra o AP P .

Figura 2.9: P representando a cadeia intermediária 01A1A0

A seguir vai uma descrição informal de P .

1. Coloque o símbolo marcador $\$$ e a variável inicial na pilha.
2. Repita os seguintes passos para sempre.
 - a. Se o topo da pilha é um símbolo de variável A , não-deterministicamente selecione uma das regras para A e substitua A pela cadeia no lado direito da regra.
 - b. Se o topo da pilha é um símbolo terminal a , leia o próximo símbolo da entrada e compare-o com a . Se eles casam, repita. Se eles não casam, rejete nesse ramo do não-determinismo.
 - c. Se o topo da pilha é o símbolo $\$$, entre no estado de aceitação. Fazendo isso aceite a cadeia se ela foi toda lida.

Prova. Agora damos os detalhes formais da construção do autômato a pilha $P = (Q, \Sigma, \Gamma, \delta, q_1, F)$. Para tornar a construção mais clara usamos notação abreviada para a função de transição. Essa notação provê uma maneira de escrever uma cadeia inteira na pilha em um passo da máquina. Podemos simular essa ação introduzindo estados adicionais para escrever a cadeia um símbolo por vez, como implementado na seguinte construção formal.

Sejam q e r estados do AP, e suponha que a esteja em Σ_ε e s esteja em Γ_ε . Digamos que desejamos que o AP vá de q para r quando ele lê a e desempilha s . Além do mais desejamos que ele empilhe a cadeia inteira $u = u_1 \dots u_l$ na pilha ao mesmo tempo. Podemos implementar essa ação introduzindo novos estados q_1, \dots, q_{l-1} e montando a função de transição da seguinte forma

$$\begin{aligned}
& \delta(q, a, s) \text{ deve conter } (q_1, u_l), \\
& \delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{l-1})\}, \\
& \delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{l-2})\}, \\
& \quad \vdots \\
& \delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}.
\end{aligned}$$

Usamos a notação $(r, u) \in \delta(q, a, s)$ para representar que quando q é o estado do autômato, a é o próximo símbolo, e s é o símbolo no topo da pilha, o AP pode ler o a e desempilhar o s , então empilhamos a cadeia u na pilha e continuar para o estado r . A Figura 2.10 mostra essa implementação pictorialmente.

Figura 2.10: Implementando a abreviação $(r, xyz) \in \delta(q, a, s)$

Os estados de P são $Q = \{q_{\text{inicio}}, q_{\text{laco}}, q_{\text{aceita}}\} \cup E$ onde E , é o conjunto de estados de que precisamos para implementar a abreviação que acaba de ser descrita. O estado inicial é q_{inicio} . O único estado de aceitação é q_{aceita} .

A função de transição é definida da seguinte forma. Começamos inicializando a pilha para conter os símbolos $\$$ e S , implementando o passo 1 na descrição informal: $\delta(q_{\text{inicio}}, \varepsilon, \varepsilon) = \{(q_{\text{laco}}, S\$)\}$. Então colocamos transições para o laço principal do passo 2.

Primeiro, lidamos com o caso (a) no qual o topo da pilha contém uma variável. Faça $\delta(q_{\text{laco}}, \varepsilon, A) = \{(q_{\text{laco}}, w) \mid \text{onde } A \rightarrow w \text{ é uma regra em } R\}$.

Segundo, lidamos com o caso (b) no qual o topo da pilha contém um terminal. Faça $\delta(q_{\text{laco}}, a, a) = \{(q_{\text{laco}}, \varepsilon)\}$.

Finalmente, lidamos com o caso (c) no qual o marcador de pilha vazia $\$$ está no topo da pilha. Faça $\delta(q_{\text{laco}}, \varepsilon, \$) = \{(q_{\text{aceita}}, \varepsilon)\}$.

O diagrama de estados é mostrado na Figura 2.11.

Figura 2.11: Diagrama de estados de P

Exemplo 2.14

Usamos o procedimento desenvolvido no Lema 2.13 para construir um AP P_1 a partir da seguinte GLC G .

$$\begin{array}{lcl}
S & \rightarrow & aTb \mid b \\
T & \rightarrow & Ta \mid \varepsilon
\end{array}$$

A função de transição é mostrada na Figura 2.12.

Figura 2.12: Diagrama de estados de P_1

Agora provamos a direção reversa do Teorema 2.12. Para a direção da ida demos um procedimento para converter um GLC em um AP. A idéia principal era projetar o autômato de modo que ele simula a gramática. Agora desejamos dar um procedimento para ir na outra direção: converter um AP em uma GLC. Projetamos a gramática para simular o autômato. Essa tarefa é um pouco complicada porque “programar” um autômato é mais fácil que “programar” uma gramática.

Lema 2.15

Se um autômato a pilha reconhece uma dada linguagem, então ela é livre-do-contexto.

.....

Idéia da prova. Temos um AP P , e desejamos construir uma GLC G que gera todas as cadeias que P aceita. Em outras palavras, G deve gerar uma cadeia se aquela cadeia faz o AP ir do seu estado inicial para um estado de aceitação.

Para atingir esse resultado projetamos uma gramática que faz algo mais. Para cada par de estados p e q em P a gramática terá uma variável A_{pq} . Essa variável gera todas as cadeias que pode levar P de p com uma pilha vazia a q com uma pilha vazia. Observe que tais cadeias podem também levar P de p para q , independente do conteúdo da pilha em p , deixando a pilha em q na mesma condição em que ela estava em p .

Primeiro, simplificamos nossa tarefa modificando P levemente para lhe dar as seguintes características.

1. Ele tem um único estado de aceitação, q_{aceita} .
2. Ele esvazia sua pilha antes de aceitar.
3. Cada transição ou empilha um símbolo na pilha (um movimento de *empilhar*) ou desempilha um da pilha (um movimento de *desempilhar*), mas não faz ambos ao mesmo tempo.

Dar a P as características 1 e 2 é fácil. Para lhe dar a característica 3, substituímos cada transição que simultaneamente desempilha e empilha com uma seqüência de duas transições que passa por um novo estado, e substituímos cada transição que nem empilha nem desempilha por uma seqüência de duas transições que empilha e depois desempilha um símbolo de pilha arbitrário.

Para projetar G de modo que A_{pq} gere todas as cadeias que levam P de p para q , começando e terminando com uma pilha vazia, temos que entender como P opera sobre essas cadeias. Para cada tal cadeia x , o primeiro movimento de P sobre x tem que ser um empilha, pois todo movimento é um empilha ou um desempilha e P não pode desempilhar uma pilha vazia. Igualmente o último movimento sobre x tem que ser um desempilha, pois a pilha termina vazia.

Duas possibilidades ocorrem durante a computação de P sobre x . Ou o símbolo desempilhado no final é o símbolo que foi empilhado no início, ou não. Se for, a pilha está vazia somente no início e no fim da computação de P sobre x . Se não for, o símbolo inicialmente empilhado tem que ser empilhado em algum ponto antes do final de x e portanto a pilha se torna vazia nesse ponto. Simulamos a primeira possibilidade com a regra $A_{pq} \rightarrow aA_{rs}b$ onde a é o símbolo de entrada lido no primeiro movimento, b é o símbolo lido no último movimento, r é o estado seguinte a p , e s o estado que precede q . Simulamos a segunda possibilidade com a regra $A_{pq} \rightarrow A_{pr}A_{rq}$, onde r é o estado no qual a pilha se torna vazia.

Prova. Digamos que $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{aceita}\})$ e vamos construir G . As variáveis de G são $\{A_{pq} \mid p, q \in Q\}$. A variável inicial é $A_{q_0, q_{aceita}}$. Agora descrevemos as regras de G .

- Para cada $p, q, r, s \in Q, t \in \Gamma$, e $a, b \in \Sigma_\varepsilon$, se $\delta(p, a, \varepsilon)$ contém (r, t) e $\delta(s, b, t)$ contém (q, ε) ponha a regra $A_{pq} \rightarrow aA_{rs}b$ em G .
- Para cada $p, q, r \in Q$ ponha a regra $A_{pq} \rightarrow A_{pr}A_{rq}$ em G .
- Finalmente, para cada $p \in Q$ ponha a regra $A_{pp} \rightarrow \varepsilon$ em G .

Você pode ganhar alguma intuição para essa construção a partir das Figuras 2.13 e 2.14.

Figura 2.13: A computação de um AP correspondendo à regra $A_{pq} \rightarrow A_{pr}A_{rq}$

Figura 2.14: A computação de um AP correspondendo à regra $A_{pq} \rightarrow aA_{rs}b$

Agora provamos que essa construção funciona demonstrando que A_{pq} gera x se e somente se (sse) x pode trazer P de p com a pilha vazia para q com a pilha vazia. Consideramos cada direção do sse como uma afirmação separada.

Afirmação 2.16

Se A_{pq} gera x , então x pode trazer P de p com a pilha vazia para q com a pilha vazia.

Provamos essa afirmação por indução sobre o número de passos na derivação de x a partir de A_{pq} .

Base: A derivação tem 1 passo.

A derivação com um único passo tem que usar uma regra cujo lado direito não contém variáveis. As únicas regras em G nas quais nenhuma variável ocorre no lado direito são $A_{pp} \rightarrow \varepsilon$. Claramente, a entrada ε leva P de p com a pilha vazia para q com a pilha vazia, portanto a base está provada.

Passo da indução: Assuma que a afirmação é verdadeira para derivações de comprimento no máximo k , onde $k \geq 1$, e prove que é verdadeira para derivações de comprimento $k + 1$.

Suponha que $A \xrightarrow{*} aA_{rs}b$ com $k + 1$ passos. O primeiro passo nessa derivação é ou $A_{pq} \Rightarrow aA_{rs}b$ ou $A_{pq} \Rightarrow A_{pr}A_{rs}$. Vamos tratar esses dois casos separadamente.

No primeiro caso, considere a porção y de x que A_{rs} gera, tal que $x = ayb$. Devido ao fato de que $A \xrightarrow{*} y$ com k passos, a hipótese da indução nos diz que P pode ir de r com a pilha vazia para s com a pilha vazia. Dado que $A_{pq} \rightarrow aA_{rs}b$ é uma regra de G , $\delta(p, a, \varepsilon)$ contém (r, t) e $\delta(s, b, t)$ contém (q, ε) . Daí, se P começa em p com a pilha vazia, após ler a ele pode ir para o estado r e empilhar t na pilha. Então ler a cadeia y pode levá-lo a s e deixar t na pilha. Então após ler b ele pode ir para o estado

q e desempilhar t da pilha. Por conseguinte x pode levá-lo de p com pilha vazia para q com pilha vazia.

No segundo caso, considere as porções y e z de x que A_{pr} e A_{rs} respectivamente geram, portanto $x = yz$. Devido ao fato de que $A_{pr} \xrightarrow{*} y$ em no máximo k passos e $A_{rs} \xrightarrow{*} z$ em no máximo k passos, a hipótese de indução nos diz que y pode trazer P de p para r , e z pode trazer P de r para q , com pilhas vazias no início e no fim. Daí x pode trazê-lo de p com pilha vazia para q com pilha vazia. Isso completa o passo da indução.

Afirmiação 2.17

Se x pode trazer P de p com pilha vazia para q com pilha vazia, A_{pq} gera x .

Provamos essa afirmação por indução sobre o número de passos na computação de P que leva de p para q com pilhas vazias sobre a entrada x .

Base: A computação tem 0 passos.

Se uma computação tem 0 passos, ela começa e termina no mesmo estado, digamos p . Portanto temos que mostrar que $A_{pp} \xrightarrow{*} x$. Em 0 passos, P apenas tem tempo de ler a cadeia vazia, portanto $x = \varepsilon$. Por construção, G tem a regra $A_{pp} \rightarrow \varepsilon$, portanto a base está provada.

Passo da indução: Assuma que a afirmação é verdadeira para computações de comprimento no máximo k , onde $k \geq 0$, e prove que é verdadeira para computações de comprimento $k + 1$.

Suponha que P tenha uma computação na qual x traz p para q com pilhas vazias em $k + 1$ passos. Ou a pilha está vazia somente no início e no fim dessa computação, ou ela se torna vazia em outro lugar também.

No primeiro caso, o símbolo que é empilhado no primeiro movimento tem que ser o mesmo que o símbolo que é desempilhado no último movimento. Chame esse símbolo t . Seja a a entrada lida no primeiro movimento, b a entrada lida no último movimento, r o estado após o primeiro movimento, e s o estado antes do último movimento. Então $\delta(p, a, \varepsilon)$ contém (r, t) e $\delta(s, b, t)$ contém (q, ε) , e portanto a regra $A_{pq} \rightarrow aA_{rs}b$ está em G .

Seja y a porção de x sem a e b , de modo que $x = ayb$. A entrada y pode trazer P de r para s sem tocar o símbolo t que está na pilha e portanto P pode ir de r com a pilha vazia para s com a pilha vazia sobre a entrada y . Removemos o primeiro e o último passos dos $k + 1$ passos na computação original sobre x portanto a computação sobre y tem $(k + 1) - 2 = k - 1$ passos. Por conseguinte a hipótese de indução nos diz que $A_{rs} \xrightarrow{*} y$. De onde $A_{pq} \xrightarrow{*} x$.

No segundo caso, seja r um estado no qual a pilha se torna vazia, e diferente do início ou do final da computação sobre x . Então as porções da computação de p para r e de r para q cada uma contém no máximo k passos. Digamos que y é a entrada lida durante a primeira porção e z é a entrada lida durante a segunda porção. A hipótese de indução nos diz que $A_{pr} \xrightarrow{*} y$ e $A_{rq} \xrightarrow{*} z$. Devido ao fato de que a regra $A_{pq} \rightarrow A_{pr}A_{rq}$ está em G , $A_{pq} \xrightarrow{*} x$, e a prova está completa.

Isto completa a prova do Lema 2.15 e do Teorema 2.12.

2.3. LINGUAGENS NÃO-LIVRES-DO-CONTEXTO 101

Acabamos de provar que autômatos a pilha reconhecem a classe das linguagens livres-do-contexto. Essa prova nos permite estabelecer um relacionamento entre as linguagens regulares e as linguagens livres-do-contexto. Devido ao fato de que toda linguagem regular é reconhecida por um autômato finito e todo autômato finito é automaticamente um autômato a pilha que simplesmente ignora sua pilha, agora sabemos que toda linguagem regular é também uma linguagem livre-do-contexto.

Corolário 2.18
Toda linguagem regular é livre-do-contexto.

Figura 2.15: Relacionamento entre linguagens regulares e linguagens livres-do-contexto

2.3 Linguagens não-livres-do-contexto

Nesta seção apresentamos uma técnica para provar que certas linguagens não são livres-do-contexto. Lembre-se que na Seção 1.4 introduzimos o lema do bombeamento para mostrar que certas linguagens não são regulares. Aqui apresentamos um lema do bombeamento semelhante para linguagens livres-do-contexto. Ele enuncia que toda linguagem livre-do-contexto tem um valor especial chamado o **comprimento de bombeamento** tal que todas as cadeias de comprimento maior que esse valor na linguagem podem ser “bombeadas.” Dessa vez o significado de *bombeada* é um pouco mais complicado. Significa que a cadeia pode ser dividida em cinco partes de modo que a segunda e a quarta partes podem ser repetidas um número qualquer de vezes e a cadeia resultante ainda permanece na linguagem.

O lema do bombeamento para linguagens livres-do-contexto

Teorema 2.19
Lema do bombeamento para linguagens livres-do-contexto Se A é uma linguagem livre-do-contexto, então existe um número p (o comprimento de bombeamento) onde, se s é uma cadeia qualquer de A de comprimento pelo menos p , então s pode ser dividida em cinco partes $s = uvxyz$ satisfazendo as condições:

1. Para cada $i \geq 0$, $uv^i xy^i z \in A$,
2. $|vy| > 0$, e
3. $|vxy| \leq p$.

Quando s está sendo dividida em $uvxyz$, a condição 2 diz que v ou y não é a cadeia vazia. Do contrário o teorema seria trivialmente verdadeiro. A condição 3 enuncia que as partes v , x , e y juntas têm comprimento no máximo p . Essa condição técnica às vezes é útil ao provar que certas linguagens são não-livres-do-contexto.

.....

Idéia da prova. Seja A uma LLC e suponha que G seja uma GLC que a gera. Temos que mostrar que qualquer cadeia suficientemente longa s em A pode ser bombeada e permanecer em A . A idéia por trás dessa abordagem é simples.

Seja s uma cadeia muito longa em A . (Esclarecemos mais adiante o que queremos dizer por “muito longa.”) Devido ao fato de que s está em A , ela é derivável de G e portanto tem uma árvore sintática. A árvore sintática para s tem que ser muito alta porque s é muito longa. Ou seja, a árvore sintática tem que conter algum caminho longo da variável inicial na raiz da árvore para um dos símbolos terminais em uma folha. Sobre esse caminho longo algum símbolo de variável R deve se repetir devido ao princípio da casa-de-pombos. Como a Figura 2.16 mostra essa repetição nos permite substituir a subárvore embaixo da segunda ocorrência de R pela subárvore embaixo da primeira ocorrência de R e ainda obter uma árvore sintática legal. Por conseguinte podemos cortar s em cinco pedaços $uvxyz$ como a figura indica, e podemos repetir o segundo e o quarto pedaços e obter uma cadeia ainda na linguagem. Em outras palavras, $uv^i xy^i z$ está em A para qualquer $i \geq 0$.

Figura 2.16: Cirurgia em árvores sintáticas

Vamos agora voltar para os detalhes para se obter todas as três condições do lema do bombeamento. Mostramos também como calcular o comprimento de bombeamento p .

Prova. Seja G uma GLC a LLC A . Seja b o número máximo de símbolos no lado de direito de uma regra. Podemos assumir que $b \geq 2$. Em qualquer árvore sintática usando essa gramática sabemos que um nó não pode ter mais que b filhos. Em outras palavras no máximo b folhas estão 1 passo da variável inicial; no máximo b^2 folhas estão a no máximo 2 passos da variável inicial; e no máximo b^h folhas estão a no máximo h passos da variável inicial. Portanto, se a altura da árvore sintática é no máximo h , o comprimento da cadeia gerada é no máximo b^h .

Seja $|V|$ o número de variáveis em G . Fazemos p ser $b^{|V|+2}$. Devido ao fato de que $b \geq 2$, sabemos que $p > b^{|V|+1}$, portanto uma árvore sintática para qualquer cadeia em A de comprimento pelo menos p requer altura no mínimo $|V| + 2$.

Suponha que s seja uma cadeia em A de comprimento no mínimo p . Agora mostramos como bombear s . Seja τ a árvore sintática para s . Se s tem várias árvores sintáticas, escolhemos τ como sendo a árvore sintática que tem o menor número de nós. Como $|s| \geq p$, sabemos que τ tem altura no mínimo $|V| + 2$, portanto o caminho mais longo em τ tem comprimento pelo menos $|V| + 2$. Esse caminho tem que ter pelo menos $|V| + 1$ variáveis porque somente a folha é um terminal. Com G tendo somente $|V|$ variáveis, alguma variável R aparece mais que uma vez no caminho. Por conveniência mais adiante, selecionamos R como sendo a variável que se repete entre as $|V| + 1$ variáveis mais baixas nesse caminho.

Dividimos s em $uvxyz$ conforme a Figura 2.16. Cada ocorrência de R tem uma subárvore sob ela, gerando uma parte da cadeia s . A ocorrência mais superior de R tem uma subárvore maior e gera vxy , enquanto que a ocorrência mais inferior gera somente x com uma subárvore menor. Ambas essas subárvores são geradas pela mesma variável, portanto podemos substituir uma pela outra e ainda assim obter uma árvore

sintática válida. Substituindo a menor pela maior repetidamente dá árvores sintáticas para as cadeias $uv^i xy^i z$ a cada $i > 1$. Substituindo a maior pela menor gera a cadeia uxz . Isso estabelece a condição 1 do lema. Agora voltamos para as condições 2 e 3.

Para obter a condição 2, temos que assegurar que ambas v e y não são ε . Se fossem, a árvore sintática obtida substituindo-se a menor subárvore pela maior teria menos nós que τ tem e ainda geraria s . Esse resultado não é possível porque já tínhamos escolhido τ como sendo uma árvore sintática para s com o menor número de nós. Essa é a razão para se selecionar τ dessa maneira.

De modo a obter a condição 3 precisamos nos assegurar de que vxy tem comprimento no máximo p . Na árvore sintática para s a ocorrência mais superior de R gera vxy . Escolhemos R de modo que ambas as ocorrências estejam entre as $|V| + 1$ variáveis de baixo no caminho, e escolhemos o caminho mais longo na árvore sintática, portanto a subárvore onde R gera vxy tem altura no máximo $|V| + 2$. Uma árvore dessa altura pode gerar uma cadeia de comprimento no máximo $b^{|V|+2} = p$.

.....

Para algumas dicas sobre o uso do lema do bombeamento para provar que linguagens são não-livres-do-contexto, reveja a página ?? onde discutimos o problema relacionado de se provar não-regularidade com o lema do bombeamento para linguagens regulares.

Exemplo 2.20

Use o lema do bombeamento para mostrar que a linguagem $B = \{a^n b^n c^n \mid n \geq 0\}$ é não-livre-do-contexto.

Assumimos que B é uma LLC e obtemos uma contradição. Seja p o comprimento de bombeamento para B que é garantido existir pelo lema do bombeamento. Selecione a cadeia $s = a^p b^p c^p$. Claramente s é um membro de B e de comprimento pelo menos p . O lema do bombeamento enuncia que s pode ser bombeada, mas mostramos que ela não pode. Em outras palavras, mostramos que independentemente de como dividimos s em $uvxyz$, uma das três condições do lema é violada.

Primeiro, a condição 2 estipula que ou v ou y seja não-vazia. Então consideramos um dos dois casos, dependendo se subcadeias v e y contêm mais que um tipo de símbolo do alfabeto.

1. Quando tanto v quanto y contêm somente um tipo de símbolos do alfabeto, v não contém ambos a 's e b 's ou ambos b 's e c 's, e o mesmo se verifica para y . Nesse caso a cadeia uv^2xy^2z não pode conter igual número de a 's, b 's, e c 's. Por conseguinte ela não pode ser um membro de B . Isso viola a condição 1 do lema e é portanto uma contradição.
2. Quando v ou y contêm mais que um tipo de símbolo uv^2xy^2z pode conter iguais números dos três símbolos do alfabeto mas não contê-los-á na ordem correta. Daí ela não pode ser um membro de B e uma contradição ocorre.

Um desses casos tem que ocorrer. Devido ao fato de que ambos os casos resultam numa contradição, uma contradição é inevitável. Portanto a suposição de que B é uma LLC tem que ser falsa. Por conseguinte provamos que B não é uma LLC.

Exemplo 2.21

Seja $C = \{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$. Usamos o lema do bombeamento para mostrar que C não é uma LLC. Essa linguagem é semelhante à linguagem B no Exemplo 2.20, mas provar que ela não é livre-do-contexto é um pouco mais complicado.

Assuma que C é uma LLC e obtenha uma contradição. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Usamos a cadeia $a^p b^p c^p$ que usamos anteriormente, mas dessa vez temos que “bombar para baixo” assim como “bombar para cima.” Seja $s = uvxyz$ e novamente considere os dois casos que ocorreram no Exemplo 2.20.

1. Quando ambas v e y contêm somente um tipo de símbolo do alfabeto, v não contém ambos a 's e b 's ou ambos b 's e c 's, e o mesmo se verifica para y . Note que o raciocínio usado previamente no caso 1 não mais se aplica. A razão é que C contém cadeias com números diferentes de a 's, b 's, e c 's desde que os números não sejam decrescentes. Temos que analisar a situação mais cuidadosamente para mostrar que s não pode ser bombeada. Observe que devido ao fato de que v e y contêm somente um tipo de símbolo do alfabeto, um dos símbolos a , b , ou c não aparece em v ou y . Dividimos mais ainda esse caso em três subcasos de acordo com qual símbolo não aparece.
 - a. Os a 's não aparecem. Então tentamos bombar para baixo para obter a cadeia $uv^0xy^0z = uxz$. Essa contém o mesmo número de a 's que s contém, mas contém menos b 's ou menos c 's. Por conseguinte ela não é um membro de C , e uma contradição ocorre.
 - b. Os b 's não aparecem. Então ou a 's ou c 's têm que aparecer em v ou y porque não pode ser que ambos sejam a cadeia vazia. Se a 's aparecem, a cadeia uv^2xy^2z contém mais a 's que b 's, portanto ela não está em C . Se c 's aparecem, a cadeia uv^0xy^0z contém mais b 's que c 's, portanto ela não está em C . Em qualquer dos casos, uma contradição ocorre.
 - c. Os c 's não aparecem. Então a cadeia uv^2xy^2z contém mais a 's ou mais b 's que c 's, portanto ela não está em C , e uma contradição ocorre.
2. Quando v ou y contêm mais que um tipo de símbolo, uv^2xy^2z não conterá os símbolos na ordem correta. Daí ela não pode ser um membro de C , e uma contradição ocorre.

Por conseguinte mostramos que s não pode ser bombeada o que viola o lema do bombeamento e que C não é livre-do-contesto.

Exemplo 2.22

Seja $D = \{ww \mid w \in \{0,1\}^*\}$. Use o lema do bombeamento para mostrar que D não é uma LLC. Assuma que D é uma LLC e obtenha uma contradição. Seja p o comprimento de bombeamento dado pelo lema do bombeamento.

Dessa vez escolher a cadeia s é menos óbvio. Uma possibilidade é a cadeia $0^p 1^p 0^p 1^p$. Ela é um membro de D e tem comprimento maior que p , portanto ela parece ser uma boa candidata. Mas essa cadeia *pode* ser bombeada dividindo-a da seguinte forma, de modo que ela não é adequada para nossos propósitos.

$$\overbrace{000 \cdots 000}^{0^p 1} \underbrace{0}_{v} \underbrace{1}_{x} \underbrace{0}_{y} \overbrace{000 \cdots 0001}^{0^p 1}$$

Vamos tentar uma outra candidata para s . Intuitivamente, a cadeia $0^p 1^p 0^p 1^p$ parece capturar mais da “essência” da linguagem D que a candidata anterior. Na verdade, podemos mostrar que essa cadeia não funciona, como segue.

Mostramos que a cadeia $s = 0^p 1^p 0^p 1^p$ não pode ser bombeada. Dessa vez usamos a condição 3 do lema do bombeamento para restringir a maneira pela qual s pode ser dividida. Ela diz que podemos bombear $s = uvxyz$, onde $|vxy| \leq p$.

Primeiro, mostramos que a subcadeia vxy tem que passar do meio da cadeia s . Caso contrário, se a cadeia ocorre somente na primeira metade de s , bombeando s para cima até uv^2xy^2z move um 1 para a primeira posição da segunda metade, e portanto ela não pode ser da forma ww . Igualmente, se vxy ocorre na segunda metade de s , bombeando s para cima até uv^2xy^2z move um 0 para a última posição da primeira metade, e portanto ela não pode ser da forma ww .

Mas se a subcadeia vxy passa do meio da cadeia s , quando tentamos bombear s para baixo até uxz ela tem a forma $0^p 1^i 0^j 1^p$, onde i e j não podem ser ambos p . Essa cadeia não é da forma ww . Por conseguinte s não pode ser bombeada, e D não é uma LLC.

.....

Exercícios

2.1 Retomemos a GLC G_4 que demos no Exemplo 2.3. Por conveniência, vamos renomear suas variáveis com letras da seguinte forma:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \times F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

Dê árvores sintáticas e derivações para cada cadeia:

- a. a
 - b. a+a
 - c. a+a+a
 - d. ((a))
- 2.2**
- a. Use as linguagens $A = \{a^m b^n c^n \mid m, n \geq 0\}$ e $B = \{a^n b^n c^m \mid m, n \geq 0\}$ juntamente com o Exemplo 2.20 para mostrar que a classe das linguagens livres-do-contexos não é fechada sob interseção.
 - b. Use a parte (a) e a lei de De Morgan (Teorema 0.10) para mostrar que a classe das linguagens livres-do-contexos não é fechada sob complementação.

2.3 Responda cada item para a seguinte gramática livre-do-contexos G :

$$\begin{aligned} R &\rightarrow XRX \mid S \\ S &\rightarrow aTb \mid bTa \\ T &\rightarrow XTX \mid X \mid \epsilon \\ X &\rightarrow a \mid b \end{aligned}$$

- a. Quais são as variáveis e terminais de G ? Qual é o símbolo inicial?
- b. Dê três exemplos de cadeias em $L(G)$.
- c. Dê três exemplos de cadeias que *não estão* em $L(G)$.
- d. Verdadeiro ou Falso: $T \Rightarrow aba$.

- e. Verdadeiro ou Falso: $T \xrightarrow{*} aba$.
- f. Verdadeiro ou Falso: $T \Rightarrow T$.
- g. Verdadeiro ou Falso: $T \xrightarrow{*} T$.
- h. Verdadeiro ou Falso: $XXX \xrightarrow{*} aba$.
- i. Verdadeiro ou Falso: $X \xrightarrow{*} aba$.
- j. Verdadeiro ou Falso: $T \xrightarrow{*} XX$.
- k. Verdadeiro ou Falso: $T \xrightarrow{*} XXX$.
- l. Verdadeiro ou Falso: $S \xrightarrow{*} \varepsilon$.
- m. Dê uma descrição em português de $L(G)$.

2.4 Dê gramáticas livres-do-contexto que geram as seguintes linguagens. Em todos os itens o alfabeto Σ é $\{0, 1\}$.

- a. $\{w \mid w \text{ contém pelo menos três } 1's\}$
- b. $\{w \mid w \text{ começa e termina com o mesmo símbolo}\}$
- c. $\{w \mid w \text{ o comprimento de } w \text{ é ímpar}\}$
- d. $\{w \mid w \text{ o comprimento de } w \text{ é ímpar e o símbolo do meio é um } 0\}$
- e. $\{w \mid w \text{ contém mais } 1's \text{ que } 0's\}$
- f. $\{w \mid w = w^R, \text{ ou seja, } w \text{ é uma palíndrome}\}$
- g. O conjunto vazio

2.5 Dê descrições informais e diagramas de estado de autômatos a pilha para as linguagens do Exercício 2.4.

2.6 Dê gramáticas livres-do-contexto gerando as seguintes linguagens:

- a. O conjunto de cadeias sobre o alfabeto $\{a, b\}$ com duas vezes mais a's que b's.
- b. O complemento da linguagem $\{a^n b^n \mid n \geq 0\}$.
- c. $\{w \# x \mid w^R \text{ é uma subcadeia de } x \text{ para } w, x \in \{0, 1\}^*\}$.
- d. $\{x_1 \# x_2 \# \dots \# x_k \mid k \geq 1, \text{ cada } x_i \in \{a, b\}^*, \text{ e para algum } i \text{ e } j, x_i = x_j^R\}$.

2.7 Dê descrições informais em português de AP's para as linguagens do Exercício 2.6.

2.8 Mostre que a cadeia `the girl touches the boy with the flower` tem duas derivações diferentes na gramática G_2 da página ???. Descreva em português os dois significados diferentes dessa sentença.

2.9 Dê uma gramática livre-do-contexto que gera a linguagem

$$A = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ e } i = j \text{ ou } j = k\}.$$

Sua gramática é ambígua? Por que ou por que não?

2.10 Dê uma descrição informal de um autômato a pilha que reconhece a linguagem A do Exercício 2.9.

2.11 Converta a GLC G_4 dada no Exercício 2.1 para um AP equivalente usando o procedimento dado no Teorema 2.12

2.12 Converta a GLC G dada no Exercício 2.3 para um AP equivalente usando o procedimento dado no Teorema 2.12

2.13 seja $G = (V, \Sigma, R, S)$ a seguinte gramática. $V = \{S, T, U\}$; $\Sigma = \{0, \#\}$; e R é o conjunto de regras abaixo:

$$\begin{aligned} S &\rightarrow TT \mid U \\ T &\rightarrow 0T \mid T0 \mid \# \\ U &\rightarrow 0U00 \mid \# \end{aligned}$$

- a. Descreva $L(G)$ em português.
- b. Prove que $L(G)$ não é regular.

2.14 Converta a seguinte GLC em uma GCL equivalente na forma normal de Chomsky, usando o procedimento dado no Teorema 2.6.

$$\begin{aligned} A &\rightarrow BAB \mid B\varepsilon \\ B &\rightarrow 00 \mid \varepsilon \end{aligned}$$

Problemas

2.15 Mostre que a classe das linguagens livres-do-contexto é fechada sob as operações regulares, união, concatenação, e estrela.

2.16 Use o resultado do Problema 2.15 para dar uma outra prova de que toda linguagem regular é livre-do-contexto, mostrando como converter uma expressão regular diretamente para uma gramática livre-do-contexto.

2.17

- a. Seja C uma linguagem livre-do-contexto e R uma linguagem regular. Prove que a linguagem $C \cap R$ é livre-do-contexto.
- b. Use o item (a) para mostrar que a linguagem $A = \{w \mid w \in \{a, b, c\}^*\text{ e contém igual número de } a\text{'s, } b\text{'s, e } c\text{'s}\}$ não é uma LLC.

2.18 Use o lema do bombeamento para mostrar que as seguintes linguagens não são livres-do-contexto.

- a. $\{0^n 1^n 0^n 1^n \mid n \geq 0\}$
- b. $\{0^n \# 0^{2n} \# 0^{3n} \mid n \geq 0\}$
- c. $\{w\#x \mid w \text{ é uma subcadeia de } x, \text{ onde } w, x \in \{a, b\}^*\}$.
- d. $\{x_1\#x_2\#\cdots\#x_k \mid k \geq 2, \text{ cada } x_i \in \{a, b\}^*, \text{ e para algum } i \neq j, x_i = x_j\}$.

2.19 Mostre que, se G é uma GLC na forma normal de Chomsky, então para qualquer cadeia $w \in L(G)$ de comprimento $n \geq 1$, exatamente $2n - 1$ passos são necessários para qualquer derivação de w .

2.20 Seja G uma GLC na forma normal de Chomsky que contém b variáveis. Mostre que, se G gera alguma cadeia usando uma derivação com pelo menos 2^b passos, $L(G)$ é infinita.

*2.21 Seja $G = (V, \Sigma, R, \langle \text{STMT} \rangle)$ a seguinte gramática:

$$\begin{aligned}\langle \text{STMT} \rangle &\rightarrow \langle \text{ASSIGN} \rangle \mid \langle \text{IF-THEN} \rangle \mid \langle \text{IF-THEN-ELSE} \rangle \mid \\ &\quad \langle \text{BEGIN-END} \rangle \\ \langle \text{IF-THEN} \rangle &\rightarrow \text{if condição then } \langle \text{STMT} \rangle \\ \langle \text{IF-THEN-ELSE} \rangle &\rightarrow \text{if condição then } \langle \text{STMT} \rangle \text{ else } \langle \text{STMT} \rangle \\ \langle \text{BEGIN-END} \rangle &\rightarrow \text{begin } \langle \text{STMT-LIST} \rangle \text{ end} \\ \langle \text{STMT-LIST} \rangle &\rightarrow \langle \text{STMT-LIST} \rangle \langle \text{STMT} \rangle \mid \langle \text{STMT} \rangle \\ \langle \text{ASSIGN} \rangle &\rightarrow \text{a := 1}\end{aligned}$$

$$\Sigma = \{\text{if, condição, then, else, begin, end, a := 1}\}.$$

$$V = \{\langle \text{STMT} \rangle, \langle \text{IF-THEN} \rangle, \langle \text{IF-THEN-ELSE} \rangle, \langle \text{BEGIN-END} \rangle, \langle \text{STMT-LIST} \rangle, \langle \text{ASSIGN} \rangle\}$$

G é uma gramática de aparência natural para um fragmento de uma linguagem de programação, mas G é ambígua.

- a. Mostre que G é ambígua.
- b. Dê uma nova gramática não-ambígua para a mesma linguagem.

2.22 Considere a linguagem $B = L(G)$, onde G é a gramática dada no Exercício 2.13. O lema do bombeamento para linguagens livres-do-contexto, Teorema 2.19, enumera a existência de um comprimento de bombeamento p para B . Qual é o valor mínimo de p que funciona no lema do bombeamento? Justifique sua resposta.

2.23 Dê um exemplo de uma linguagem que não é livre-do-contexto mas que satisfaz as três condições do lema do bombeamento. Prove que seu exemplo funciona. (Veja o fato análogo para linguagens regulares no Problema 1.37.)

*2.24 Mostre que a linguagem A do Exercício 2.9 é inherentemente ambígua.

*2.25 Seja G a GLC

$$\begin{aligned}S &\rightarrow aSb \mid bY \mid Ya \\ Y &\rightarrow bY \mid aY \mid \varepsilon\end{aligned}$$

Dê uma descrição simples de $L(G)$ em português. Use essa descrição para dar uma GLC para $\overline{L(G)}$, o complemento de $L(G)$.

2.26 Seja $C = \{x\#y \mid x, y \in \{0, 1\}^\text{ e }x \neq y\}$. Mostre que C é uma linguagem livre-do-contexto.

2.27 Seja $D = \{xy \mid x, y \in \{0, 1\}^\text{ e }|x| = |y|\text{ mas }x \neq y\}$. Mostre que C é uma linguagem livre-do-contexto.

*2.28 Prove a seguinte forma mais forte do lema do bombeamento, no qual requeremos que *ambas* as partes v e y sejam não-vazias quando a cadeia s é quebrada.

Se A é uma linguagem livre-do-contexto, então existe um número k onde, se s é uma cadeia qualquer em A de comprimento pelo menos k , então s pode ser dividida em cinco partes, $s = uvxyz$, satisfazendo as condições:

- a. Para cada $i \geq 0$, $uv^i xy^i z \in A$,
- b. $v \neq \varepsilon$ e $y \neq \varepsilon$, e
- c. $|vxy| \leq k$.

Parte Dois: Teoria da Computabilidade

Capítulo 3

A Tese de Church–Turing

Até agora no nosso desenvolvimento da teoria da computação apresentamos vários modelos de dispositivos de computação. Autômatos finitos são bons modelos para dispositivos que têm uma quantidade pequena de memória. Autômatos a pilha são bons modelos para dispositivos que têm uma quantidade de memória ilimitada que é utilizável apenas da forma “o último que entra é o primeiro que sai” tal qual uma pilha. Mostramos que algumas tarefas muito simples estão além das capacidades desses modelos. Portanto eles são demasiado restritos para servir como modelos de computadores de propósito geral.

3.1 Máquinas de Turing

Voltamo-nos agora para um modelo muito mais poderoso, primeiramente proposto por Alan Turing em 1936, chamado de *máquinas de Turing*. Semelhante a um autômato finito mas com uma memória ilimitada e irrestrita, uma máquina de Turing é um modelo muito mais preciso de um computador de propósito geral. Uma máquina de Turing pode fazer tudo o que um computador real pode fazer. Entretanto, mesmo uma máquina de Turing não pode resolver certos problemas. Num sentido muito real, esses problemas estão além dos limites teóricos da computação.

O modelo da máquina de Turing usa uma fita infinita como sua memória ilimitada. Ela tem uma cabeça que pode ler e escrever símbolos e mover sobre a fita. Inicialmente a fita contém somente a cadeia de entrada e está em branco em todo o restante. Se a máquina precisa de armazenar informação, ela pode escrever essa informação sobre a fita. Para ler a informação que ela escreveu, a máquina pode mover sua cabeça de volta sobre a posição onde a informação foi escrita. A máquina continua a computar até que ela decide produzir uma saída. As saídas *aceita* e *rejeita* são obtidas pela entrada em estados de aceitação e de rejeição. Se ela não entrar em estado de aceitação ou de rejeição, ela continuará para sempre, nunca parando.

Figura 3.1: Esquema de uma máquina de Turing

A seguinte lista resume as diferenças entre autômatos finitos e máquinas de Turing.

1. Uma máquina de Turing pode tanto escrever sobre a fita quanto ler dela.
2. A cabeça de leitura-escrita pode mover tanto para a esquerda quanto para a direita.
3. A fita é infinita.
4. Os estados especiais para rejeitar e aceitar têm efeito imediato.

Vamos considerar uma máquina de Turing M_1 para testar pertinência na linguagem $B = \{w\#w \mid w \in \{0, 1\}^*\}$. Isto é, desejamos projetar M_1 para aceitar se sua entrada é um membro de B . Para entender M_1 melhor, ponha-se no seu lugar imaginando que você está sobre uma entrada de 1Km de comprimento consistindo de milhões de caracteres. Seu objetivo é determinar se a entrada é um membro de B , ou seja, se a entrada compreende duas cadeias idênticas separadas por um símbolo $\#$. A entrada é demasiado longa para você memorizá-la toda, mas lhe é permitido mover de-frente-para-trás e de-trás-para-frente sobre a entrada e deixar marcas sobre ela. É claro que a estratégia óbvia é ziguezaguear para as posições correspondentes nos dois lados do $\#$ e determinar se eles casam. Use marcas para manter o registro de quais posições se correspondem.

Projetamos M_1 para funcionar da mesma maneira. Ela realiza múltiplas varridas sobre a cadeia de entrada com a cabeça de leitura-escrita. A cada passagem ela emparelha um dos caracteres em cada lado do símbolo $\#$. Para manter registro de quais símbolos já foram verificados, M_1 deixa uma marca sobre cada símbolo à medida que ele é examinado. Se ela marca todos os símbolos, isso significa que tudo emparelhou de forma bem sucedida, e M_1 vai para um estado de aceitação. Se ela descobre uma sobra, ela entra em um estado de rejeição. Em resumo, o algoritmo de M_1 é o seguinte.

M_1 = “Sobre a cadeia de entrada w :

1. Faça um varredura na entrada para assegurar que ela contém uma única ocorrência do símbolo $\#$. Se não, *rejeite*.
2. Faça um zigue-zague na fita para fazer corresponder posições nos dois lados do símbolo $\#$ para verificar se essas posições contêm o mesmo símbolo. Se elas não contêm, *rejeite*. Marque os símbolos à medida que eles são verificados para manter registro de quais símbolos têm correspondência.
3. Quando todos os símbolos à esquerda do $\#$ foram marcados, verifique se existe algum símbolo remanescente à direita do $\#$. Se quaisquer símbolos restam, *rejeite*; caso contrário, *aceite*.

A Figura 3.2 contém várias fotografias instantâneas da fita de M_1 enquanto ela está computando nos estágios 2 e 3 quando iniciada sobre a entrada 011000#011000.

Figura 3.2: Fotografias instantâneas da máquina de Turing M_1 computando sobre a entrada 011000#011000

Essa descrição da máquina de Turing M_1 esboça a maneira como ela funciona mas não dá todos os detalhes. Podemos descrever máquinas de Turing em todos os detalhes dando descrições formais análogas àquelas introduzidas para autômatos finitos e autômatos a pilha. A descrição formal especifica cada uma das partes da definição formal do modelo da máquina de Turing a ser apresentada logo mais. Na realidade quase nunca damos descrições formais de máquinas de Turing porque elas tendem a ser muito grandes.

Definição formal de uma máquina de Turing

O coração da definição de uma máquina de Turing é a função de transição δ porque ela diz como a máquina vai de um passo ao próximo. Para uma máquina de Turing, δ toma a forma $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{E, D\}$. Ou seja, quando a máquina está num certo estado q e a cabeça está sobre uma posição da fita contendo um símbolo a , e se $\delta(q, a) = (r, b, E)$, a máquina escreve o símbolo b substituindo o a , e vai para o estado r . O terceiro componente é E ou D e indica se a cabeça move para a esquerda ou para a direita após a escrita. Nesse caso, o E indica um movimento para a esquerda.

Definição 3.1

Uma **máquina de Turing** é uma 7-upla $(Q, \Sigma, \Gamma, \delta, q_0, q_{aceita}, q_{rejeita})$, onde Q, Σ, Γ são todos conjuntos finitos e

1. Q é o conjunto de estados,
2. Σ é o alfabeto de entrada que não contém o símbolo especial **branco** \sqcup ,
3. Γ é o alfabeto da fita, onde $\sqcup \in \Gamma$ e $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{E, D\}$ é a função de transição,
5. q_0 é o estado inicial,
6. $q_{aceita} \in Q$ é o estado de aceitação,
7. $q_{rejeita} \in Q$ é o estado de rejeição.

Uma máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{aceita}, q_{rejeita})$ computa da seguinte forma. Inicialmente M recebe sua entrada $w = w_1 w_2 \dots w_n \in \Sigma^*$ nas células mais à esquerda da fita, e o restante da fita é branco (i.e., preenchido com símbolos em branco). A cabeça começa na célula mais à esquerda da fita. Note que Σ não contém o símbolo branco, portanto o símbolo branco aparecendo na fita marca o final da entrada. Uma vez que M começa, a computação prossegue conforme as regras descritas pela função de transição. Se M em algum momento tenta mover sua cabeça para à esquerda do final à esquerda da fita, a cabeça permanece no mesmo lugar para aquele movimento, muito embora a função de transição indique E . A computação continua até que ela entre nos estados de aceitação ou rejeição nos quais ela pára. Se nenhum desses casos ocorre, M continua para sempre.

À medida que uma máquina de Turing computa, mudanças ocorrem no estado atual, no conteúdo da fita, e na localização da cabeça. Uma combinação dessas três informações é chamada uma **configuração** da máquina de Turing. Configurações frequentemente são representadas de uma maneira especial. Para um estado q e duas cadeias u e v sobre o alfabeto da fita Γ escrevemos uqv para a configuração onde o

estado atual é q , o conteúdo da fita é uv , e a localização atual da cabeça é o primeiro símbolo de v . A fita contém apenas brancos a partir do último símbolo de v . Por exemplo, $1011q_701111$ representa a configuração quando a fita é 10110111 , o estado atual é q_7 , e a cabeça está atualmente sobre o segundo 0. A Figura 3.3 mostra uma máquina de Turing com aquela configuração.

Figura 3.3: Uma máquina de Turing com configuração $1011q_701111$

Aqui formalizamos nosso entendimento intuitivo da maneira pela qual uma máquina de Turing computa. Vamos dizer que uma configuração C_1 **produz** uma configuração C_2 se a máquina de Turing pode legalmente ir de C_1 para C_2 em um único passo. Definimos essa noção formalmente da seguinte maneira.

Suponha que tenhamos a, b , e c em Γ , assim como u e v em Γ^* e estados q_i e q_j . Nesse caso uaq_ibv e uq_jacv são duas configurações. Vamos dizer que

$$uaq_ibv \text{ produz } uq_jacv$$

se na função de transição $\delta(q_i, b) = (q_j, c, E)$. Isso lida com o caso em que a máquina de Turing move para a esquerda. Para um movimento à direita, vamos dizer que

$$uaq_ibv \text{ produz } uacq_jv$$

se $\delta(q_i, b) = (q_j, c, D)$.

Casos especiais ocorrem quando a cabeça está em uma das extremidades da configuração. Para a extremidade esquerda, a configuração $q_i bv$ produz $q_j cv$ se a transição é de movimento para a esquerda (porque cuidamos para que a máquina não passe da extremidade à esquerda da fita), e ela produz $c q_j v$ para a transição de movimento à direita. Para a extremidade direita, a configuração $ua q_i$ é equivalente a $ua q_i \sqcup$ porque assumimos que brancos vêm logo depois da parte da fita representada na configuração. Por conseguinte podemos lidar com esse caso tal qual anteriormente, com a cabeça não mais na extremidade direita.

A **configuração inicial** de M sobre a entrada w é a configuração $q_0 w$, que indica que a máquina está no estado inicial q_0 com sua cabeça na posição mais à esquerda sobre a fita. Em uma **configuração de aceitação** o estado da configuração é q_{aceita} . Em uma **configuração de rejeição** o estado da configuração é q_{rejeita} . Configurações de aceitação e de rejeição são **configurações de parada** e como tal não produzem quaisquer configurações. Uma máquina de Turing M **aceita** a entrada w se uma seqüência de configurações C_1, C_2, \dots, C_k existe onde

1. C_1 é a configuração inicial de M sobre a entrada w ,
2. cada C_i produz C_{i+1} , e
3. C_k é uma configuração de aceitação.

A coleção de cadeias que M aceita é a **linguagem de M** , denotada $L(M)$.

Definição 3.2

Chame uma linguagem **Turing-reconhecível** se alguma máquina de Turing a reconhece.¹

Quando iniciamos uma MT sobre uma entrada, três resultados são possíveis. A máquina pode *aceitar*, *rejeitar*, ou *entrar em loop*. Por *entrar em loop* queremos dizer que a máquina simplesmente não pára. Ela não está necessariamente repetindo os mesmos passos para sempre como a conotação de entrar em loop pode sugerir. Entrar em loop pode acarretar comportamento simples ou complexo que nunca leva a um estado de parada.

Uma máquina de Turing M pode falhar em aceitar uma cadeia entrando no estado *rejeita* e rejeitando, ou entrando em loop. Às vezes distinguir uma máquina que está em loop de uma que está meramente levando um tempo longo é difícil. Por essa razão preferimos máquinas de Turing que parem sobre todas as entradas; tais máquinas nunca entram em loop. Essas máquinas são chamadas **decisores** porque elas sempre tomam uma decisão de aceitar ou de rejeitar. Um decisor que reconhece uma linguagem também é dito **decidir** aquela linguagem.

Definição 3.3

Chame uma linguagem **Turing-decidível** ou simplesmente **decidível** se alguma máquina de Turing a decide.²

Toda linguagem decidível é Turing-reconhecível mas certas linguagens Turing-reconhecíveis não são decidíveis. Agora damos alguns exemplos de linguagens decidíveis. Apresentamos exemplos de linguagens que são Turing-reconhecíveis mas não decidíveis após desenvolvêrmos uma técnica para provar indecidibilidade no Capítulo 4.

Exemplos de máquinas de Turing

Como fizemos para autômatos finitos e autômatos a pilha, podemos dar uma descrição formal de uma máquina de Turing particular especificando cada uma das suas sete partes. Entretanto, indo para aquele nível de detalhe para máquinas de Turing pode ser incômodo para todas exceto as máquinas minúsculas. Assim, não desperdiçaremos muito tempo dando tais descrições. No máximo daremos apenas descrições de mais alto nível porque elas são suficientemente precisas para nossos propósitos e são muito mais fáceis de entender. Entretanto, é importante lembrar que toda descrição de mais alto nível é na verdade simplesmente uma abreviação da sua contrapartida formal. Com paciência e cuidado poderíamos descrever qualquer das máquinas de Turing neste livro em detalhe formal completo.

Para ajudar você a fazer a conexão entre as descrições formais e as descrições de mais alto nível, damos diagramas de estado nos próximos dois exemplos. Você pode pulá-los se você já se sente confortável com essa conexão.

Exemplo 3.4

Aqui descrevemos uma MT M_2 que reconhece a linguagem consistindo de todas as cadeias de 0's cujo comprimento é uma potência de 2. Ela decide a linguagem $A = \{0^{2^n} \mid n \geq 0\}$.

M_2 = “Sobre a cadeia de entrada w :

¹Ela é chamada uma linguagem **recursivamente enumerável** em alguns outros livros-texto.

²Ela é chamada uma linguagem **recursiva** em alguns outros livros-texto.

1. Faça uma varredura da esquerda para a direita na fita, apagando um 0 alternadamente. (O primeiro não, o segundo sim, etc.)
2. Se no estágio 1 a fita continha um único 0, *aceite*.
3. Se no estágio 1 a fita continha mais de um único 0 e o número de 0's era ímpar, *rejeite*.
4. Retorne a cabeça para a extremidade esquerda da fita.
5. Vá para o estágio 1.”

Cada iteração do estágio 1 corta o número de 0's pela metade. À medida que a máquina faz a varredura na fita no estágio 1, ela mantém registro do fato de que o número de 0's é par ou ímpar. Se aquele número é ímpar e maior que 1, o número original de 0's na entrada não poderia ter sido uma potência de 2. Por conseguinte a máquina rejeita nesta instância. Entretanto, se o número de 0's vistos é 1, o número original tem que ter sido uma potência de 2. Portanto nesse caso a máquina aceita.

Agora damos a descrição formal de $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{aceita}}, q_{\text{rejeita}})$.

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{aceita}}, q_{\text{rejeita}}\}$,
- $\Sigma = \{0\}$, e
- $\Gamma = \{0, x, \sqcup\}$.
- Descrevemos δ com um diagrama de estados (veja Figura 3.4).
- Os estados inicial, de aceitação, e de rejeição são q_1, q_{aceita} , e q_{rejeita} .

No diagrama de estados da Figura 3.4 o rótulo $0 \rightarrow \sqcup, D$ aparece na transição de q_1 para q_2 . Significa que, quando no estado q_1 com a cabeça lendo 0, a máquina vai para o estado q_2 , escreve \sqcup , e move a cabeça para a direita. Em outras palavras, $\delta(q_1, 0) = (q_2, \sqcup, D)$. Para maior clareza usamos a abreviação $0 \rightarrow D$ na transição de q_3 para q_4 , para indicar que a máquina move para a direita quando lê 0 no estado q_3 mas não altera a fita, portanto $\delta(q_3, 0) = (q_4, 0, D)$.

Essa máquina começa escrevendo um símbolo em branco sobre o 0 mais à esquerda na fita de modo que ela possa encontrar a extremidade esquerda da fita no estágio 4. Enquanto que normalmente usaríamos um símbolo mais sugestivo tal como # para o delimitador de extremidade esquerda, usamos um branco aqui para manter o alfabeto da fita, e portanto o diagrama de estados, pequeno. O Exemplo 3.6 dá um outro método de encontrar a extremidade esquerda da fita.

Damos uma amostra de execução dessa máquina sobre a entrada 0000. A configuração inicial é $q_1 0 0 0 0$. A seqüência de configurações que a máquina entra aparece após a Figura 3.4. Leia as colunas de cima para baixo e da esquerda para a direita.

Figura 3.4: Diagrama de estados para a máquina de Turing M_2

Uma amostra de execução de M_2 sobre a entrada 0000:

$q_1 0000$	$\sqcup q_5 x 0 x \sqcup$	$\sqcup x q_5 x x \sqcup$
$\sqcup q_2 000$	$q_5 \sqcup x 0 x \sqcup$	$\sqcup q_5 x x x \sqcup$
$\sqcup x q_3 00$	$\sqcup q_2 x 0 x \sqcup$	$q_5 \sqcup x x x \sqcup$
$\sqcup x 0 q_4 0$	$\sqcup x q_2 x 0 \sqcup$	$\sqcup q_2 x x x \sqcup$
$\sqcup x 0 x q_3 \sqcup$	$\sqcup x x q_3 x \sqcup$	$\sqcup x q_2 x x \sqcup$
$\sqcup x 0 q_5 x \sqcup$	$\sqcup x x x q_3 \sqcup$	$\sqcup x x q_2 x \sqcup$
$\sqcup x q_5 0 x \sqcup$	$\sqcup x x q_5 x \sqcup$	$\sqcup x x x q_2 \sqcup$
		$\sqcup x x x \sqcup q_{\text{aceita}}$

Exemplo 3.5

A seguir vai uma descrição formal de $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{aceita}}, q_{\text{rejeita}})$, a máquina de Turing que informalmente descrevemos na página 127 para decidir a linguagem $B = \{w\#w \mid w \in \{0, 1\}^*\}$.

1. $Q = \{q_1, \dots, q_{14}, q_{\text{aceita}}, q_{\text{rejeita}}\}$,
2. $\Sigma = \{0, 1, \#\}$, e $\Gamma = \{0, 1, \#, x, \sqcup\}$.
3. Descrevemos δ com um diagrama de estados (veja Figura 3.5).
4. Os estados inicial, de aceitação, e de rejeição são $q_1, q_{\text{aceita}}, q_{\text{rejeita}}$.

Na Figura 3.5 que ilustra o diagrama de estados da MT M_1 , você vai encontrar o rótulo $0, 1 \rightarrow D$ sobre a transição indo de q_3 para si próprio. Esse rótulo significa que a máquina permanece em q_3 e move para a direita quando ela lê um 0 ou um 1 no estado q_3 . Ela não muda o símbolo sobre a fita.

Figura 3.5: Diagrama de estados para a máquina de Turing M_1

Como no Exemplo 3.4, a máquina começa escrevendo um símbolo em branco para delimitar a fronteira à esquerda da fita. Dessa vez ela pode escrever um 0 ou um 1 por cima quando fizer isso, e memorizar o símbolo escrito por cima usando um controle finito.

O estágio 1 é implementado pelos estados q_1 até q_7 , e os estágios 2 e 3 pelos estados remanescentes. Para simplificar a figura, não mostramos o estado de rejeição ou as transições indo para o estado de rejeição. Essas transições ocorrem implicitamente sempre que um estado não dispõe de uma transição saindo dele para um símbolo específico. Por conseguinte, devido ao fato de que no estado q_5 nenhuma seta de saída com um $\#$ está presente, se um $\#$ ocorre sob a cabeça de leitura-escrita quando a máquina está no estado q_5 , ela vai para o estado q_{rejeita} .

Exemplo 3.6

Aqui, uma máquina de Turing M_3 está fazendo aritmética elementar. Ela decide a linguagem $C = \{a^i b^j c^k \mid i \times j = k \text{ e } i, j, k \geq 1\}$.

M = “Sobre a cadeia de entrada w :

1. Faça uma varredura na entrada da esquerda para a direita para se assegurar que ela é um membro de $a^*b^*c^*$ e *rejeite* se ela não for.
2. Retorne a cabeça para a extremidade esquerda da fita.
3. Marque um a e faça uma varredura à esquerda até que um b ocorra. Viaje entre os b 's e os c 's, apagando um de cada até que todos os b 's tenham se acabado.
4. Restaure os b 's marcados e repita o estágio 3 se existe um outro a para marcar. Se todos os a 's estão marcados, verifique se todos os c 's também estão marcados. Se sim, *aceite*; caso contrário, *rejeite*.”

Vamos examinar os quatro estágios de M_3 mais de perto: No estágio 1 a máquina opera como um autômato finito. Nenhuma escrita é necessária quando a máquina move da esquerda para a direita, mantendo registro, por meio do uso de seus estados, se a entrada está na forma apropriada.

O estágio 2 parece igualmente simples mas contém uma sutileza. Como pode a máquina de Turing encontrar a extremidade esquerda da fita de entrada? Encontrar a extremidade direita da fita é fácil porque ela está terminada com um símbolo branco. Mas a extremidade esquerda não tem terminador inicialmente. Uma técnica que permite a máquina encontrar a extremidade esquerda da fita é marcar o símbolo mais à esquerda de alguma forma quando a máquina inicia com sua cabeça sobre aquele símbolo. Então a máquina pode fazer uma varredura para a esquerda até que ela encontra a marca quando ela deseja reinicializar sua cabeça para a extremidade esquerda. O Exemplo 3.4 ilustrou essa técnica, usando um símbolo branco para marcar o símbolo à esquerda na fita.

Um método mais elaborado de encontrar a extremidade esquerda da fita aproveita-se da forma pela qual definimos o modelo da máquina de Turing. Lembre-se que, se a máquina tenta mover sua cabeça além da extremidade esquerda da fita, ela permanece no mesmo lugar. Podemos usar essa característica para montar um detector de extremidade esquerda. Para detectar se a cabeça está posicionada na extremidade esquerda a máquina pode escrever um símbolo especial sobre a posição atual, enquanto grava no controle o símbolo que ela substituiu. Então ela pode tentar mover a cabeça para a esquerda. Se ela ainda está sobre o símbolo especial, o movimento para a esquerda não foi bem sucedido, e por conseguinte a cabeça tem que ter estado sobre a extremidade esquerda. Se, ao contrário, ela está sobre um símbolo diferente, alguns símbolos permaneceram à esquerda daquela posição sobre a fita. Antes de ir adiante, a máquina tem que garantir que restaura o símbolo modificado sobre a fita para o símbolo original.

Os estágios 3 e 4 têm implementações simples usando vários estados cada.

Exemplo 3.7

Aqui, uma máquina de Turing M_4 está resolvendo o que é chamado do *problema da diferenciação de elemento*. A ela é dada uma lista de cadeias sobre $\{0, 1\}$ separadas por #'s e sua tarefa é aceitar se todas as cadeias são diferentes. A linguagem é

$$E = \{\#x_1\#x_2\#\cdots\#x_l \mid \text{cada } x_i \in \{0, 1\}^* \text{ e } x_i \neq x_j \text{ para cada } i \neq j\}.$$

A máquina M_4 funciona comparando x_1 com x_2 até x_l , e então comparando x_2 com x_3 até x_l , e assim por diante. Uma descrição informal da MT M_4 que decide essa linguagem segue.

M_4 = “Sobre a entrada w :

1. Ponha uma marca sobre o símbolo mais à esquerda. Se aquele símbolo era um branco, *aceite*. Se aquele símbolo era um #, continue com o próximo estágio. Caso contrário, *rejeite*.
2. Faça uma varredura para a direita até o próximo # e ponha uma segunda marca sobre ele. Se nenhum # for encontrado antes de um símbolo branco, somente x_1 estava presente, portanto *aceite*.
3. Fazendo zigue-zague, compare as duas cadeias à direita dos #'s marcados. Se eles são iguais, *rejeite*.
4. Mova a marca mais à direita das duas para o próximo # à direita. Se nenhum símbolo # for encontrado antes de um símbolo branco, move a marca mais à esquerda do próximo # para a sua direita e a marca mais à direita para o # após aquele. Dessa vez, se nenhum # está disponível para a marca mais à direita, todas as cadeias foram comparadas, portanto *aceite*.
5. Vá para o estágio 3.”

Essa máquina ilustra a técnica de marcar símbolos da fita. No estágio 2, a máquina põe uma marca sobre um símbolo, # nesse caso. Na implementação real, a máquina tem dois símbolos diferentes, # e \$, no seu alfabeto de fita. Dizer que uma máquina põe uma marca sobre um # significa que a máquina escreve o símbolo # naquela posição. Remover a marca significa que a máquina marca sobre vários símbolos sobre a fita. Para fazer isso simplesmente incluímos versões de todas esses símbolos da fita com pontos no alfabeto da fita.

Podemos concluir dos exemplos precedentes que as linguagens descritas A , B , C , E são decidíveis. Todas as linguagens decidíveis são Turing-reconhecíveis, portanto essas linguagens são também Turing-reconhecíveis. Demonstrar que uma linguagem é Turing-reconhecível nas não decidíveis é mais difícil, o que fazemos no Capítulo 4.

3.2 Variantes de máquinas de Turing.....

Definições alternativas de máquinas de Turing abundam, incluindo versões com múltiplas fitas ou com não-determinismo. Elas são chamadas *variantes* do modelo da máquina de Turing. O modelo original e suas variantes razoáveis todos têm o mesmo poder—eles reconhecem a mesma classe de linguagens. Nesta seção descrevemos algumas dessas variantes e as provas de equivalência em poder. Chamamos *robustez* essa invariância a certas mudanças na definição. Tanto autômatos finitos como autômatos a pilha são modelos um tanto robustos, mas máquinas de Turing têm um grau impressionante de robustez.

Para ilustrar a robustez do modelo de máquina de Turing vamos variar o tipo de função de transição permitida. Em nossa definição, a função de transição força a cabeça a mover para a esquerda ou para a direita após cada passo; a cabeça não pode simplesmente permanecer parada. Suponha que tivéssemos permitido à máquina de Turing a capacidade de permanecer parada. A função de transição teria então a forma $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{E, D, P\}$. Essa característica poderia permitir às máquinas de Turing reconhecer linguagens adicionais, e por conseguinte adicionar poder ao modelo? É claro que não, porque podemos converter qualquer MT com a característica

adicional de “permanecer parada” para uma que não tem essa característica adicional. Fazemos isso substituindo cada transição com “permanecer parada” por duas transições, uma que move para a direita e a segunda que volta para a esquerda.

Esse pequeno exemplo contém a chave para mostrar a equivalência de variantes da máquina de Turing. Para mostrar que dois modelos são equivalentes precisamos simplesmente de mostrar que podemos simular um por meio do outro.

Máquinas de Turing multi-fitas

Uma **máquina de Turing multi-fitas** é como uma máquina de Turing comum com várias fitas. Cada fita tem sua própria cabeça para ler e escrever. Inicialmente a entrada aparece sobre a fita 1, e as outras começam com branco. A função de transição é modificada para permitir ler, escrever, e mover as cabeças em todas as fitas simultaneamente. Formalmente, ela é

$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{E, D\}^k$$

onde k é o número de fitas. A expressão

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, E, D, \dots, E)$$

quer dizer que, se a máquina está no estado q_i e a cabeça 1 até a cabeça k estão lendo os símbolos a_1 até a_k , a máquina vai para o estado q_j , escreve os símbolos b_1 até b_k , e move cada cabeça para a esquerda ou a direita conforme especificado.

Máquinas de Turing multi-fitas parecem ser mais poderosas que máquinas de Turing comuns, mas podemos mostrar que elas são equivalentes em poder. Lembre-se que duas máquinas são equivalentes se elas reconhecem a mesma linguagem.

Teorema 3.8

Toda máquina de Turing multi-fitas tem uma máquina de Turing com uma única fita equivalente.

Prova. Mostramos como converter uma MT multi-fitas M em uma MT S com uma única fita equivalente. A idéia chave é mostrar como simular M com S .

Digamos que M tem k fitas. Então S simula o efeito de k fitas armazenando a informação que elas trazem numa única fita. Ela usa o novo símbolo $\#$ como um delimitador para separar o conteúdo das diferentes fitas. Além do conteúdo dessas fitas, S tem que manter registro das posições das cabeças. Ela faz isso escrevendo um símbolo de fita com um ponto acima dele para marcar o lugar onde a cabeça sobre aquela fita estaria. Pense como se fossem fitas e cabeças “virtuais”. Tal qual anteriormente, os símbolos com um ponto em cima são simplesmente novos símbolos que foram adicionados ao alfabeto de fita. A Figura 3.6 ilustra como uma fita pode ser usada para representar três fitas.

Figura 3.6: Representando três fitas com uma

S = “Sobre a entrada $w = w_1 \dots w_n$:

3.2. VARIANTES DE MÁQUINAS DE TURING 121

1. Primeiro S põe sua fita no formato que representa todas as k fitas de M . A fita formatada contém

$$\# \dot{w}_1 w_2 \cdots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$$

2. Para simular um único movimento, S faz uma varredura na sua fita a partir do primeiro $\#$, que marca a extremidade esquerda, até o $(k+1)$ -ésimo $\#$, que marca a extremidade direita, de modo a determinar os símbolos sob as cabeças virtuais. Então S faz uma segunda varredura para atualizar as fitas conforme a maneira pela qual a função de transição de M determina.
3. Se em qualquer ponto S move uma das cabeças virtuais para a direita sobre um $\#$, essa ação significa que M moveu a cabeça correspondente sobre a parte em branco ainda não lida da fita. Portanto S escreve um símbolo branco sobre essa célula da fita e desloca o conteúdo da fita, a partir dessa célula até o $\#$ mais à direita, uma unidade para a direita. E aí continua a simulação como antes.”

Corolário 3.9

Uma linguagem é Turing-reconhecível se e somente se alguma máquina de Turing multi-fitas a reconhece.

Prova. Uma linguagem Turing-reconhecível é reconhecida por uma máquina de Turing comum (com uma única fita), o que é um caso especial de uma máquina de Turing multi-fitas. Isso prova uma direção deste corolário. A outra direção segue do Teorema 3.8.

Máquinas de Turing não-determinísticas

Uma máquina de Turing não-determinística é definida da maneira esperada. Em qualquer ponto em uma computação a máquina pode proceder conforme várias possibilidades. A função de transição para uma máquina de Turing não-determinística tem a forma

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{E, D\}).$$

A computação de uma máquina de Turing não-determinística é uma árvore cujos ramos correspondem a possibilidades diferentes para a máquina. Se algum ramo da computação leva ao estado de aceitação, a máquina aceita sua entrada. Se você sente a necessidade de revisar não-determinismo, volte para a Seção 1.2 na página 47. Agora mostramos que não-determinismo não afeta o poder do modelo de máquina de Turing.

Teorema 3.10

Toda máquina de Turing não-determinística tem uma máquina de Turing determinística equivalente.

Idéia da Prova. Mostramos que podemos simular qualquer MT não-determinística N com uma MT determinística D . A idéia por trás da simulação é fazer com que D tente todos os possíveis ramos da computação não-determinística de N . Se D chega a encontrar o estado de aceitação sobre um desses ramos, D aceita. Caso contrário, a simulação de D não termina.

Vemos a computação de N sobre uma entrada w como uma árvore. Cada ramo da árvore representa um dos ramos do não-determinismo. Cada nó da árvore é uma configuração de N . A raiz da árvore é a configuração inicial. A MT D faz uma busca nessa árvore por uma configuração de aceitação. Conduzir essa busca cuidadosamente é crucial para que D não deixe de visitar a árvore inteira. Uma idéia tentadora, embora ruim, é fazer com que D explore a árvore usando busca em profundidade. A estratégia da busca em profundidade vai até o fundo num ramo antes de voltar e explorar outros ramos. Se D tivesse que explorar a árvore dessa maneira, D poderia descer para sempre num ramo infinito e perder uma configuração de aceitação em algum outro ramo. Daí projetamos D para explorar a árvore usando busca em largura ao invés de busca em profundidade. Essa estratégia explora todos os ramos até a mesma profundidade antes de continuar a explorar qualquer ramo na próxima profundidade. Esse método garante que D visitará todo nó na árvore até que ela encontre uma configuração de aceitação.

Prova. A MT D simuladora tem três fitas. Pelo Teorema 3.8 esse arranjo é equivalente a ter uma única fita. A máquina D usa suas três fitas de uma maneira particular, conforme ilustrado na Figura 3.7. A fita 1 sempre contém a cadeia de entrada e nunca é alterada. A fita 2 mantém uma cópia da fita de N sobre algum ramo de sua computação não-determinística. A fita 3 mantém registro da posição de D na árvore de computação não-determinística de N .

Figura 3.7: MT determinística D simulando a MT não-determinística N

Vamos primeiro considerar a representação de dados na fita 3. Todo nó na árvore pode ter no máximo b filhos, onde b é o tamanho do maior conjunto de escolhas possíveis dado pela função de transição de N . A todo nó na árvore atribuimos um endereço que é uma cadeia sobre o alfabeto $\Sigma_b = \{1, 2, \dots, b\}$. Atribuimos o endereço 231 ao nó a que chegamos iniciando na raiz, indo para seu 2º filho, indo para o 3º daquele nó, e finalmente indo para o 1º filho daquele nó. Cada símbolo na cadeia diz qual escolha a fazer a seguir quando se simula um passo num ramo da computação não-determinística de N . Às vezes um símbolo pode não corresponder a nenhuma escolha se muito poucas escolhas estão disponíveis para uma configuração. Nesse caso o endereço é inválido e não corresponde a nenhum nó. A fita 3 contém uma cadeia sobre Σ_b . Ela representa o ramo da computação de N a partir da raiz para o nó endereçado por aquela cadeia, a menos que o endereço seja inválido. A cadeia vazia é o endereço da raiz da árvore. Agora estamos prontos para descrever D .

1. Inicialmente a fita 1 contém a cadeia w , e as fitas 2 e 3 estão vazias.
2. Copie a fita 1 para a fita 2.
3. Use a fita 2 para simular N com entrada w sobre um ramo de sua computação não-determinística. Antes de cada passo de N consulte o próximo símbolo sobre a fita 3 para determinar qual escolha fazer entre aquelas permitidas pela função de transição de N . Se nenhum símbolo mais permanece sobre a fita 3 ou se esse escolha não-determinística for inválida, aborte esse ramo indo para o estágio 4. Também vá para o estágio 4 se uma configuração de rejeição for encontrada. Se uma configuração de aceitação for encontrada, *aceite* a entrada.

3.2. VARIANTES DE MÁQUINAS DE TURING 123

4. Substitua a cadeia sobre a fita 3 pela cadeia lexicograficamente seguinte. Simule o próximo ramo da computação de N indo para o estágio 2.

Corolário 3.11

Uma linguagem é Turing-reconhecível se e somente se alguma máquina de Turing não-determinística a reconhece.

Prova. Qualquer MT determinística é automaticamente uma MT não-determinística e portanto uma direção desse teorema segue imediatamente. A outra direção segue do Teorema 3.10.

.....
Podemos modificar a prova do Teorema 3.10 de modo que se N sempre pára em todos os ramos de sua computação, D sempre vai parar. Chamamos uma máquina de Turing não-determinística um **decisor** se todos os ramos param sobre todas as entradas. O Exercício 3.3 pede a você para modificar a prova dessa maneira para obter o seguinte corolário do Teorema 3.10.

Corolário 3.12

Uma linguagem é decidível se e somente se alguma máquina de Turing não-determinística a decide.

Enumeradores

Como mencionamos em uma nota de pé-de-página anterior, algumas pessoas usam o termo linguagem **recursivamente enumerável** para uma linguagem Turing-reconhecível. Esse termo tem origem num tipo de variante de máquina de Turing chamado um enumerador. Frouxamente definido, um enumerador é uma máquina de Turing com uma impressora em anexo. A máquina de Turing pode usar essa impressora como um dispositivo de saída para imprimir cadeias. Toda vez que a máquina de Turing quer adicionar uma cadeia à lista, ela envia a cadeia para a impressora. O Exercício 3.4 pede que você dê uma definição formal de um enumerador. A Figura 3.8 ilustra um esquema desse modelo.

Figura 3.8: Esquema de um enumerador

Um enumerador começa com uma fita de entrada em branco. Se o enumerador não pára, ele pode imprimir uma lista infinita de cadeias. A linguagem enumerada por E é a coleção de todas as cadeias que ela eventualmente imprime. Além do mais, E pode gerar as cadeias da linguagem em qualquer ordem, possivelmente com repetições. Agora estamos prontos para desenvolver a conexão entre enumeradores e linguagens Turing-reconhecíveis.

Teorema 3.13

Uma linguagem é Turing-reconhecível se e somente se algum enumerador a enumera.

Prova. Primeiro mostramos que se temos um enumerador E que enumera uma linguagem A , uma MT M reconhece A . A MT M funciona da seguinte forma.

M = “Sobre a entrada w :

1. Rode E . Toda vez que E dá como saída uma cadeia, compare-a com w .
2. Se w vem a aparecer na saída de E , *aceite*.”

Claramente M aceita aquelas cadeias que aparecem na lista de E .

Agora fazemos a outra direção. Se a MT M reconhece uma linguagem A , podemos construir o seguinte enumerador E para A . Digamos que s_1, s_2, \dots é uma lista de todas as possíveis cadeias em Σ^* .

E = “Ignore a entrada.

1. Repita o seguinte para $i = 1, 2, 3, \dots$
2. Rode M por i passos sobre cada entrada s_1, s_2, \dots, s_i .
3. Se quaisquer computações aceitam, imprima o s_j correspondente.”

Se M aceita uma cadeia específica s , eventualmente ela aparecerá na lista gerada por E . Na verdade, ela aparecerá na lista uma quantidade infinita de vezes porque M roda do início sobre cada cadeia para cada repetição do passo 1. Esse procedimento dá o efeito de se rodar M em paralelo sobre todas as possíveis cadeias de entrada.

.....

Equivalência com outros modelos

Até aqui apresentamos diversas variantes do modelo da máquina de Turing e demonstramos que elas são equivalentes em poder. Muitos outros modelos são muito parecidos com máquinas de Turing, enquanto outros são um tanto diferentes. Todos compartilham a característica essencial de máquinas de Turing, a saber, acesso irrestrito a memória ilimitada, distingindo-os de modelos mais fracos tais como autômatos finitos e autômatos a pilha. Notavelmente, *todos* os modelos com essa característica acabam sendo equivalentes em poder, desde que eles satisfaçam certos requisitos razoáveis.³

Para entender esse fenômeno considere a situação análoga para linguagens de programação. Muitas, tais como Pascal e LISP, parecem um tanto diferentes umas das outras em estilo e estrutura. Algum algoritmo pode ser programado em uma delas e não nas outras? Claro que não—podemos compilar LISP para Pascal e Pascal para LISP, o que significa que as duas linguagens descrevem *exatamente* a mesma classe de algoritmos. Assim o fazem todas as outras linguagens de programação razoáveis. A tão difundida equivalência de modelos computacionais se verifica precisamente pela mesma razão. Quaisquer modelos computacionais que satisfaçam certos requisitos razoáveis podem simular um ao outro e portanto são equivalentes em poder.

Esse fenômeno da equivalência tem um corolário filosófico importante. Muito embora existam muitos modelos computacionais diferentes, a classe de algoritmos que eles descrevem é única. Enquanto que cada modelo computacional individual tem um certo grau de arbitrariedade na sua definição, a classe subjacente de algoritmos que ele descreve é natural porque ela é a mesma classe que outros modelos descrevem. Esse fenômeno também tem tido profundas implicações para a matemática, como mostramos na próxima seção.

³Por exemplo, um requisito é a capacidade de realizar somente uma quantidade finita de trabalho em um único passo.

3.3 A definição de algoritmo

Informalmente falando, um *algoritmo* é uma coleção de instruções simples para levar adiante alguma tarefa. Comum na vida do dia-a-dia, algoritmos às vezes são chamados *procedimentos ou receitas*. Algoritmos também desempenham um papel importante em matemática. A antiga literatura matemática contém descrições de algoritmos para uma variedade de tarefas, tais como encontrar números primos e máximos divisores comuns. Na matemática contemporânea algoritmos abundam.

Muito embora algoritmos tenham tido uma longa história em matemática, a noção de algoritmo propriamente dita não estava definida precisamente até o século XX. Antes disso, os matemáticos tinham uma noção intuitiva do que algoritmos eram e se apoiavam naquela noção ao usá-los e descrevê-los. Mas aquela noção intuitiva era insuficiente para adquirir um entendimento mais profundo de algoritmos. A estória a seguir relata como a definição precisa de algoritmo foi crucial para um importante problema matemático.

Os problemas de Hilbert

Em 1900, o matemático David Hilbert proferiu uma hoje-famosa palestra no Congresso Internacional de Matemáticos em Paris. Em sua apresentação, ele identificou vinte-e-três problemas matemáticos e os colocou como um desafio para o século vindouro. O décimo problema na sua lista dizia respeito a algoritmos.

Antes de descrever aquele problema, vamos discutir brevemente polinômios. Um *polinômio* é uma soma de termos, onde cada *termo* é um produto de certas variáveis e uma constante chamada um *coeficiente*. Por exemplo

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

é um termo com coeficiente 6, e

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

é um polinômio com quatro termos sobre as variáveis x , y , e z . Uma *raiz* de um polinômio é uma atribuição de valores a suas variáveis tal que o valor do polinômio seja 0. Esse polinômio tem uma raiz em $x = 5$, $y = 3$, e $z = 0$. Essa raiz é uma *raiz inteira* porque todas as variáveis são substituídas por valores inteiros. Alguns polinômios têm uma raiz inteira e alguns não.

O décimo problema de Hilbert era para projetar um algoritmo que testa se um polinômio tem uma raiz inteira. Ele não usou o termo *algoritmo* mas sim “um processo conforme o qual ela pode ser determinada por um número finito de operações.”⁴ É interessante observar que, da forma com que ele fraseou esse problema, Hilbert pedia explicitamente que um algoritmo fosse “projetado.” Por conseguinte ele aparentemente assumiu que tal algoritmo tinha que existir—alguém só precisa encontrá-lo.

Como agora sabemos, nenhum algoritmo existe para essa tarefa; ela é algorítmicamente insolúvel. Para os matemáticos daquele período chegar a essa conclusão com seu conceito intuitivo de algoritmo teria sido virtualmente impossível. O conceito intuitivo pode ter sido adequado para dar algoritmos para certas tarefas, mas ele era inútil para mostrar que nenhum algoritmo existe para uma tarefa particular. Provar que um algoritmo não existe requer dispor de uma definição clara de algoritmo. Progresso no décimo problema teve que esperar por aquela definição.

⁴Traduzido do original em alemão.

A definição veio nos artigos de 1936 de Alonzo Church e Alan Turing. Church usou um sistema notacional chamado o λ -cálculo para definir algoritmos. Turing o fez com suas “máquinas.” Essas duas definições foram demonstradas serem equivalentes. Essa conexão veio a ser chamada a *tese de Church-Turing*.

A tese de Church-Turing provê a definição de algoritmo necessária para resolver o décimo problema de Hilbert. Em 1970, Yuri Matijasevič, baseando-se em trabalhos de Martin Davis, Hilary Putnam, e Julia Robinson, mostrou que nenhum algoritmo existe para testar se um polinômio tem raízes inteiras. No Capítulo 4 desenvolvemos as técnicas que formam a base para provar que esse e outros problemas são algoritmamente insolúveis.

<i>Noção intuitiva de algoritmo</i>	<i>é igual a</i>	<i>Algoritmos de máquinas de Turing</i>
-----------------------------------------	------------------	---------------------------------------------

Figura 3.9: A Tese de Church-Turing

Vamos frasear o décimo problema de Hilbert em nossa terminologia. Fazer isso ajuda a introduzir alguns temas que exploramos nos Capítulos 4 e 5. Seja

$$D = \{p \mid p \text{ é um polinômio com uma raiz inteira}\}.$$

O décimo problema de Hilbert pergunta em essência se o conjunto D é decidível. A resposta é negativa. Por outro lado podemos mostrar que D é Turing-reconhecível. Antes de fazer isso, vamos considerar um problema mais simples. É um análogo ao décimo problema de Hilbert para polinômios que têm apenas uma única variável, tais como $4x^3 - 2x^2 + x - 7$. Seja

$$D_1 = \{p \mid p \text{ é um polinômio sobre } x \text{ com uma raiz inteira}\}.$$

Aqui está uma máquina de Turing M_1 que reconhece D_1 :

$M =$ “A entrada é um polinômio p sobre a variável x .

1. Calcule o valor de p com x substituído sucessivamente pelos valores $0, 1, -1, 2, -2, 3, -3, \dots$ Se em algum ponto o polinômio resulta no valor 0, *aceite*.”

Se p tem uma raiz inteira, M_1 eventualmente a encontrará e aceitará. Se p não tem uma raiz inteira, M_1 vai rodar para sempre. Para o caso multivariado, podemos apresentar uma máquina de Turing semelhante M que reconhece D . Aqui, M passar por todas as possíveis atribuições de valor a suas variáveis com valores inteiros.

Ambas M_1 e M são reconhecedores mas não decisores. Podemos converter M_1 para um decisor para D_1 porque podemos calcular limitantes dentro dos quais as raízes de uma única variável polinomial tem que residir e restringir a busca a esses limitantes. No Problema 3.18 lhe será pedido para mostrar que as raízes de tal polinômio têm que estar entre os valores

$$\pm k \frac{c_{\max}}{c_1},$$

onde k é o número de termos no polinômio, c_{\max} é o coeficiente com o maior valor absoluto, e c_1 é o coeficiente do termo de mais alta ordem. se uma raiz não for encontrada dentro desses limitantes, a máquina *rejeita*. O teorema de Matijasevič mostra que calcular tais limitantes para polinômios multivariados é impossível.

Terminologia para descrever máquinas de Turing

Chegamos a um ponto de virada no estudo da teoria da computação. Continuamos a falar de máquinas de Turing, mas nosso foco real de agora em diante é sobre algoritmos. Ou seja, a máquina de Turing serve meramente como um modelo preciso para a definição de algoritmo. Pularemos a extensa teoria de máquinas de Turing propriamente ditas e não gastaremos muito tempo na programação de baixo-nível de máquinas de Turing. Precisamos apenas estar confortável o bastante com máquinas de Turing para acreditar que elas capturam todos os algoritmos.

Com isso em mente, vamos padronizar a maneira que descrevemos algoritmos de máquinas de Turing. Incialmente, perguntamos: Qual é o nível correto de detalhe a dar ao descrever tais algoritmos? Estudantes comumente fazem essa pergunta, especialmente ao preparar soluções a exercícios e problemas. Vamos entreter três possibilidades. A primeira é a *descrição formal* que explicita por completo os estados da máquina de Turing, função de transição, e assim por diante. É o nível mais baixo, mais detalhado, de descrição. A segunda é uma descrição de mais alto nível, chamada a *descrição de implementação*, na qual usamos prosa em português para descrever a maneira pela qual a máquina de Turing movimenta sua cabeça e a forma pela qual ela armazena dados na sua fita. Nesse nível não damos detalhes de estados ou função de transição. A terceira é a *descrição de alto-nível*, na qual usamos prosa em português para descrever um algoritmo, ignorando o modelo de implementação. Nesse nível não precisamos mencionar como a máquina administra sua fita ou sua cabeça.

Neste capítulo demos descrições formais e de nível-de-implementação de vários exemplos de máquinas de Turing. A prática com descrições de mais baixo nível de máquina de Turing ajuda a você entender máquinas de Turing e ganhar confiança em usá-las. Uma vez que você se sente confiante, descrições de alto-nível são suficientes.

Agora estabelecemos um formato e notação para descrever máquinas de Turing. A entrada para uma máquina de Turing é sempre uma cadeia. Se desejamos prover um objeto diferente de uma cadeia como entrada, temos primeiro que representar aquele objeto como uma cadeia. Cadeias podem facilmente representar polinômios, grafos, gramáticas, autômatos, e qualquer combinação daqueles objetos. Uma máquina de Turing pode ser programada para decodificar a representação de modo que ela pode ser interpretada da maneira que pretendemos. Nossa notação para a codificação de um objeto O em sua representação como uma cadeia é $\langle O \rangle$. Se temos vários objetos O_1, O_2, \dots, O_k , denotamos sua codificação em uma única cadeia por $\langle O_1, O_2, \dots, O_k \rangle$. A codificação propriamente dita pode ser feita de várias maneiras razoáveis. Não importa qual escolhemos, porque uma máquina de Turing pode sempre traduzir uma dessas codificações em uma outra.

Em nosso formato, descrevemos algoritmos de máquinas de Turing com um segmento indentado de texto dentro de aspas. Quebramos o algoritmo em estágios, cada um usualmente envolvendo muitos passos individuais da computação da máquina de Turing. Indicamos a estrutura de bloco do algoritmo com mais indentação. Se a descrição da entrada é simplesmente w , a entrada é formada como sendo uma cadeia. Se a descrição da entrada é a codificação de um objeto como em $\langle A \rangle$, a máquina de Turing primeiramente implicitamente testa se a entrada codifica apropriadamente um objeto para a forma desejada e a rejeita se ela não codifica.

Exemplo 3.14

Seja A a linguagem consisting de todas as cadeias representando grafos não-direcionados que são conexos. Lembre-se que um grafo é **conexo** se todo nó pode ser atingido a par-

tir de qualquer outro nó por meio de uma viagem sobre as arestas do grafo. Escrevemos

$$A = \{\langle G \rangle \mid G \text{ é um grafo não-direcionado conexo}\}.$$

A seguir vai uma descrição de alto-nível de uma MT que decide A .

M = “Sobre a entrada $\langle G \rangle$, a codificação de um grafo G :

1. Selecione o primeiro nó de G e marque-o.
2. Repita o estágio seguinte até que nenhum nó novo seja marcado.
3. Para cada nó em G , marque-o se ele está ligado por uma aresta a um nó que já está marcado.
4. Faça uma varredura em todos os nós de G para determinar se eles todos estão marcados. Se eles estão, *aceite*; caso contrário *rejeite*.”

Para prática adicional, vamos examinar alguns detalhes de nível-de-implementação da máquina de Turing M . Usualmente não daremos esse nível de detalhe no futuro e nem você precisará fazê-lo, a menos que especificamente solicitado em um exercício. Primeiro, temos que entender como $\langle G \rangle$ codifica o grafo G como uma cadeia. Considere uma codificação que é uma lista dos nós de G seguida por uma lista das arestas de G . Cada nó é um número decimal, e cada aresta é o par de números decimais que representam os nós nas duas extremidades da aresta. A Figura 3.10 ilustra esse grafo e sua codificação.

Figura 3.10: Um grafo G e sua codificação $\langle G \rangle$

Quando M recebe a entrada $\langle G \rangle$, ela primeiro procura determinar se a entrada é uma codificação apropriada de algum grafo. Para fazer isso, M varre a fita para se assegurar de que existem duas listas e que elas estão na forma apropriada. A primeira lista deve ser uma lista de números decimais distintos, e a segunda deve ser uma lista de pares de números decimais. Então M verifica várias coisas. Primeiro, a lista de nós não deve conter repetições, e segundo, todo nó aparecendo na lista de aresta deve também aparecer na lista de nós. Para o primeiro, podemos usar o procedimento dado no Exemplo 3.7 para a MT M_4 que verificar distinção de elementos. Um método semelhante funciona para o segundo item de verificação. Se w passa nessas verificações, ela é a codificação de algum grafo G . Essa verificação completa a checagem da entrada, e M vai para o estágio 1.

Para o estágio 1, M marca o primeiro nó com um ponto sobre o dígito mais à esquerda.

Para o estágio 2, M varre a lista de nós para encontrar um nó n_1 sem-ponto e o destaca marcando-o diferentemente, digamos, sublinhando o primeiro símbolo. Então M varre a lista novamente para encontrar um nó n_2 marcado com um ponto e o sublinha também.

Agora M varre a lista de arestas. Para cada aresta, M testa se os dois nós sublinhados n_1 e n_2 são aqueles aparecendo naquela aresta, podemos usar o procedimento dado no Exemplo 3.7 para a MT M_4 que verificar distinção de elementos. Um método semelhante funciona para o segundo item de verificação. Se w passa nessas verificações, ela

é a codificação de algum grafo G . Essa verificação completa a checagem da entrada, e M vai para o estágio 1.

Para o estágio 1, M marca o primeiro nó com um ponto sobre o dígito mais à esquerda.

Para o estágio 2, M varre a lista de nós para encontrar um nó n_1 sem-ponto e o destaca marcando-o diferentemente, digamos, sublinhando o primeiro símbolo. Então M varre a lista novamente para encontrar um nó n_2 marcado com um ponto e o sublinha também.

Agora M varre a lista de arestas. Para cada aresta, M testa se os dois nós sublinhados n_1 e n_2 são os que aparecem naquela aresta. Se eles o são, M marca com um ponto n_1 , remove as marcas de sublinhar, e continua a partir do início do estágio 2. Se eles não o são, M verifica a próxima aresta na lista. Se não existem mais arestas, $\{n_1, n_2\}$ não é uma aresta de G . Então M move o símbolo de sublinhar em n_2 para o próximo nó marcado com um ponto e agora chama esse nó de n_2 . Repete os passos neste parágrafo para verificar, como antes, se o novo par $\{n_1, n_2\}$ é uma aresta. Se não existem mais nós marcados com um ponto, n_1 não está ligado a nenhum nó marcado com um ponto. Então M lança os símbolos de sublinhar de tal modo que n_1 é o próximo nó não-marcado com um ponto e n_2 é o primeiro nó marcado com um ponto e repete os passos neste parágrafo. Se não existem mais nós não-marcados com um ponto, M não foi capaz de encontrar nenhum novo nó para marcar com um ponto, portanto ela vai para o estágio 4.

Para o estágio 4, M varre a lista de nós para determinar se todos estão marcados com um ponto. Se eles estão, ela entra no estado de aceitação; caso contrário ela entra no estado de rejeição. Isso completa a descrição da MT M .

Exercícios

3.1 Este exercício concerne a MT M_2 cuja descrição e diagrama de estados aparecem no Exemplo 3.4. Em cada um dos itens abaixo, dê a seqüência de configurações nas quais M_2 entra quando iniciado sobre a cadeia de entrada indicada:

- a. 0.
- b. 00.
- c. 000.
- d. 000000.

3.2 Este exercício concerne a MT M_1 cuja descrição e diagrama de estados aparecem no Exemplo 3.5. Em cada um dos itens abaixo, dê a seqüência de configurações nas quais M_1 entra quando iniciado sobre a cadeia de entrada indicada:

- a. 11.
- b. 1#1.
- c. 1##1.
- d. 10#11.
- e. 10#10.

- 3.3** Modifique a prova do Teorema 3.10 na página 138 para obter o Corolário 3.12 que mostra que uma linguagem é decidível se alguma MT não-determinística a decide. (Você pode assumir o seguinte teorema sobre árvores: Se todo nó em uma árvore tem uma quantidade finita de filhos e todo ramo da árvore tem uma quantidade finita de nós, a árvore propriamente dita tem uma quantidade finita de nós.)
- 3.4** Dê uma definição formal de um enumerador. Considere-o como sendo um tipo de máquina de Turing de duas-fitas que usa sua segunda fita como a impressora. Inclua a definição da linguagem enumerada.
- 3.5** Examine a definição formal de uma máquina de Turing para responder às seguintes perguntas, e explique seu raciocínio.
- Uma máquina de Turing pode alguma vez escrever o símbolo branco \square na sua fita?
 - O alfabeto de fita Γ pode ser o mesmo que o alfabeto de entrada Σ ?
 - A cabeça de uma máquina de Turing pode *alguma vez* estar na mesma localização em dois passos sucessivos?
 - Uma máquina de Turing pode conter apenas um único estado?

- 3.6** No Teorema 3.13 mostramos que uma linguagem é Turing-reconhecível se algum enumerador a enumera. Por que não usamos o seguinte algoritmo mais simples para a direção de-trás-para-frente da prova? Tal qual anteriormente, s_1, s_2, \dots é uma lista de todas as cadeias em Σ^* .

E = “Ignore a entrada.

- Repita o seguinte para $i = 1, 2, 3, \dots$
- Rode M sobre s_i .
- Se ela aceita, imprima s_i .“

- 3.7** Explique por que a descrição abaixo não é uma descrição de uma máquina de Turing legítima.

M_{ruim} = “A entrada é um polinômio p sobre variáveis x_1, \dots, x_k .

- Tente todas as possíveis valorações de x_1, \dots, x_k para valores inteiros.
- Calcule p sobre todas essas valorações.
- Se alguma dessas valorações resulta em 0, *aceite*; caso contrário, *rejeite*.“

- 3.8** Dê descrições a nível de implementação de máquinas de Turing que decidem as linguagens abaixo sobre o alfabeto $\{0, 1\}$.

- $\{w \mid w \text{ contém o mesmo número de } 0's \text{ e } 1's\}$.
- $\{w \mid w \text{ contém duas vezes mais } 0's \text{ que } 1's\}$.
- $\{w \mid w \text{ não contém duas vezes mais } 0's \text{ que } 1's\}$.

Problemas

3.9 Seja um k -AP um autômato a pilha que tem k pilhas. Por conseguinte um 0-AP é um AFN e um 1-AP é um AP convencional. Você já sabe que 1-AP's são mais poderosos (reconhecem uma classe maior de linguagens) que 0-AP's.

- Mostre que 2-AP's são mais poderosos que 1-AP's.
- Mostre que 3-AP's não são mais poderosos que 2-AP's.

(Dica: Simule uma fita de uma máquina de Turing com duas fitas.)

3.10 Digamos que uma *máquina de Turing de apenas-uma-escrita* é uma MT de uma única fita que pode alterar cada célula da fita no máximo uma vez (incluindo a parte ocupada pela entrada na fita). Mostre que essa variante do modelo da máquina de Turing é equivalente ao modelo comum da máquina de Turing. (Dica: Como um primeiro passo considere o caso no qual a máquina de Turing pode alterar cada célula da fita no máximo duas vezes. Use bastante fita.)

3.11 Uma *máquina de Turing com fita duplamente infinita* é semelhante a uma máquina de Turing comum exceto que sua fita é infinita para a esquerda assim como para a direita. A fita é inicialmente preenchida com brancos exceto a parte que contém a entrada. Computação é definida como de costume exceto que a cabeça nunca encontra um fim da fita quando move para a esquerda. Mostre que esse tipo de máquina de Turing reconhece a classe de linguagens Turing-reconhecíveis.

3.12 Uma *máquina de Turing com reinicialização à esquerda* é semelhante a uma máquina de Turing comum exceto que sua função de transição tem a forma

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{D, REINICIA\}.$$

Se $\delta(q, a) = (r, b, REINICIA)$, quando a máquina está no estado q lendo um a , a cabeça da máquina salta para a extremidade esquerda da fita após ela escrever b na fita e entra no estado r . Note que essas máquinas não têm a capacidade usual de mover a cabeça um símbolo à esquerda. Mostre que máquinas de Turing com reinicialização à esquerda reconhecem a classe de linguagens Turing-reconhecíveis.

3.13 Uma *máquina de Turing com movimento nulo ao invés de à esquerda* é semelhante a uma máquina de Turing comum exceto que sua função de transição tem a forma

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{D, N\}.$$

Em cada ponto a máquina pode mover sua cabeça para a direita ou permanecer na mesma posição. Mostre que essa variante da máquina de Turing não é equivalente à versão usual. Que classe de linguagens essas máquinas reconhecem?

3.14 Mostre que a coleção de linguagens decidíveis é fechada sob as operações de

- união.
- concatenação.
- estrela.

- d. complementação.
- e. interseção.

3.15 Mostre que a coleção de linguagens Turing-reconhecíveis é fechada sob as operações de

- a. união.
- b. concatenação.
- c. estrela.
- d. interseção.

***3.16** Mostre que uma linguagem é decidível se algum enumerador enumera a linguagem na ordem lexicográfica.

***3.17** Mostre que MT's de uma única fita que não podem escrever na parte da fita contendo a cadeia de entrada podem somente reconhecer linguagens regulares.

3.18 Seja $c_1x^n + c_2x^{n-1} + \dots + c_nx + c_{n+1}$ um polinômio com uma raiz em $x = x_0$. Seja c_{\max} o maior valor absoluto de um c_i . Mostre que

$$|x_0| < (n+1) \frac{c_{\max}}{|c_1|}.$$

3.19 Seja A a linguagem contendo somente uma única cadeia s , onde

$$s = \begin{cases} 0 & \text{se Deus não existe} \\ 1 & \text{se Deus existe.} \end{cases}$$

A é decidível? Por que, ou por que não? (Note que sua resposta não depende de suas convicções religiosas.)

Capítulo 4

Decidibilidade

No Capítulo 3 introduzimos a máquina de Turing como um modelo de um computador de propósito geral e definimos a noção de algoritmo em termos de máquinas de Turing por meio da tese de Church–Turing.

Neste capítulo começamos a investigar o poder de algoritmos para resolver problemas. Demonstramos certos problemas que podem ser resolvidos algorítmicamente e outros que não podem. Nossa objetivo é explorar os limites da solubilidade algorítmica. Você está provavelmente familiarizado com solubilidade por algoritmos porque muito da ciência da computação é dedicado a resolver problemas. A insolubilidade de certos problemas pode vir como uma surpresa.

Por que você deveria estudar insolubilidade? Afinal de contas, mostrar que um problema é insolúvel não parece ser de qualquer utilidade se você tem que resolvê-lo. Você precisa estudar esse fenômeno por duas razões. Primeiro, saber quando um problema é algorítmicamente insolúvel é útil porque então você se dá conta de que o problema deve ser simplificado ou alterado antes que você possa encontrar uma solução algorítmica. Como qualquer ferramenta, computadores têm capacidades e limitações que têm que ser apreciadas se elas são para serem bem usadas. A segunda razão é cultural. Mesmo se você lida com problemas que claramente são solúveis, uma olhadela no insolúvel pode estimular sua imaginação e ajudá-lo a ganhar uma importante perspectiva sobre computação.

4.1 Linguagens decidíveis.....

Nesta seção damos alguns exemplos de linguagens que são decidíveis por algoritmos. Por exemplo, apresentamos um algoritmo que testa se uma cadeia é um membro de uma linguagem livre-do-contexto. Esse problema está relacionado ao problema de reconhecer e compilar programas em uma linguagem de programação. Ver algoritmos resolvendo vários problemas concernentes a autômatos é útil, porque mais adiante você encontrará outros problemas concernentes a autômatos que não podem ser resolvidos por algoritmos.

Problemas decidíveis concernentes a linguagens regulares

Começamos com certos problemas computacionais concernentes a autômatos finitos. Damos algoritmos para testar se um autômato finito aceita uma cadeia, se a linguagem de um autômato é vazia, e se dois autômatos finitos são equivalentes.

Por conveniência usamos linguagens para representar vários problemas computacionais porque já estabelecemos terminologia para lidar com linguagens. Por exemplo, o **problema da aceitação** para AFD's de testar se um autômato finito específico aceita uma dada cadeia pode ser expresso como uma linguagem A_{AFD} . Essa linguagem contém as codificações de todos os AFD's juntamente com cadeias que os AFD's aceitam. Seja

$$A_{\text{AFD}} = \{\langle B, w \rangle \mid B \text{ é um AFD que aceita a cadeia de entrada } w\}.$$

O problema de testar se um AFD B aceita uma entrada w é o mesmo que o problema de testar se $\langle B, w \rangle$ é um membro da linguagem A_{AFD} . Igualmente, podemos formular outros problemas computacionais em termos de testar pertinência em uma linguagem. Mostrar que uma linguagem é decidível é o mesmo que mostrar que o problema computacional é decidível.

No teorema a seguir mostrarmos que A_{AFD} é decidível. Portanto esse teorema mostra que o problema de testar se um dado autômato finito aceita uma dada cadeia é decidível.

Teorema 4.1
 A_{AFD} é uma linguagem decidível.

.....
Idéia da prova. A idéia da prova é muito simples. Precisamos somente de apresentar uma MT M que decide A_{AFD} .

M = “Sobre a entrada $\langle B, w \rangle$, onde B é um AFD e w é uma cadeia:

1. Simule B sobre a entrada w .
2. Se a simulação termina em um estado de aceitação, *aceite*. Se ela termina em um estado de não-aceitação, *rejeite*.”

Prova. Mencionamos somente alguns poucos detalhes de implementação desta prova. Para aqueles de vocês familiares com escrever programas em qualquer linguagem de programação padrão, imagine como você escreveria um programa para levar adiante a simulação.

Primeiro, vamos examinar a entrada $\langle B, w \rangle$. Ela é uma representação de um AFD B juntamente com uma cadeia w . Uma representação razoável de B é simplesmente uma lista de seus cinco componentes, Q , Σ , δ , q_0 , e F . Quando M recebe sua entrada, M primeiro verifica se ela representa propriamente um AFD B e uma cadeia w . Se não, M rejeita.

Então M leva adiante a simulação de uma maneira direta. Ela mantém registro do estado atual de B e da posição atual de B sobre a entrada w escrevendo essa informação na sua fita. Inicialmente, o estado atual de B é q_0 e a posição atual de B sobre a entrada é o símbolo mais à esquerda de w . Os estados e a posição são atualizados conforme a função de transição especificada δ . Quando M termina de processar o último símbolo de w , M aceita a entrada se B está num estado de aceitação; M rejeita a entrada se B está em um estado de não-aceitação.

Podemos provar um teorema semelhante para autômatos finitos não-determinísticos. Seja

$$A_{\text{AFN}} = \{\langle B, w \rangle \mid B \text{ é um AFN que aceita a cadeia de entrada } w\}.$$

Teorema 4.2

A_{AFN} é uma linguagem decidível.

Prova. Apresentamos uma MT N que decide A_{AFN} . Poderíamos projetar N para operar como M , simulando um AFN ao invés de um AFD. Ao invés disso, faremos diferente para ilustrar uma nova idéia: fazer N usar M como uma subrotina. Devido ao fato de que M é projetado para funcionar com AFD's, N primeiro converte o AFN que ele recebe como entrada para um AFD antes de passá-lo para M .

N = “Sobre a entrada $\langle B, w \rangle$ onde B é um AFN, e w é uma cadeia:

1. Converta o AFN B para um AFD C equivalente usando o procedimento para conversão dado no Teorema 1.19.
2. Rode a MT M do Teorema 4.1 sobre a entrada $\langle C, w \rangle$.
3. Se M aceita, *aceite*; caso contrário *rejeite*.”

Rodar a MT M no estágio 2 significar incorporar M no projeto de N como um subprocedimento.

Igualmente, podemos testar se uma expressão regular gera uma dada cadeia. Seja $A_{\text{EXR}} = \{\langle R, w \rangle \mid R \text{ é uma expressão regular que gera a cadeia } w\}$.

Teorema 4.3

A_{EXR} é uma linguagem decidível.

Prova. A seguinte MT P decide A_{EXR} .

P = “Sobre a entrada $\langle R, w \rangle$ onde R é uma expressão regular e w é uma cadeia:

1. Converta a expressão regular R para um AFD equivalente A usando o procedimento para essa conversão dado no Teorema 1.28.
 2. Rode a MT M sobre a entrada $\langle A, w \rangle$.
 3. Se M aceita, *aceite*; se M rejeita, *rejeite*.”
-

Os Teoremas 4.1, 4.2, e 4.3 ilustram que para propósitos de decidibilidade, apresentar uma máquina de Turing com um AFD, um AFN, ou uma expressão regular são todos equivalentes porque a máquina é capaz de converter uma forma de codificação na outra.

Agora nos voltamos para um tipo diferente de problema concernente a autômatos finitos: **testar vacuidade** para a linguagem de um autômato finito. Nos teoremas precedentes tínhamos que testar se um autômato finito aceita uma cadeia específica. Na próxima prova temos que testar se um autômato finito aceita alguma cadeia. Seja

$$V_{\text{AFD}} = \{\langle A \rangle \mid A \text{ é um AFD e } L(A) = \emptyset\}.$$

Teorema 4.4

V_{AFD} é uma linguagem decidível.

Prova. Um AFD aceita alguma cadeia se e somente se atingir um estado de aceitação a partir do estado inicial viajando por setas do AFD é possível. Para testar essa condição podemos projetar uma MT T que usa um algoritmo de marcação semelhante àquele usado no Exemplo 3.14.

T = “Sobre a entrada $\langle A \rangle$ onde A é um AFD:

1. Marque o estado inicial de A .
 2. Repita até que nenhum estado novo venha a ser marcado:
 3. Marque qualquer estado que tem uma transição chegando nele a partir de qualquer estado que já esteja marcado.
 4. Se nenhum estado de aceitação está marcado, *aceite*; caso contrário *rejeite*.“
-

O próximo teorema enuncia que testar se dois AFD’s reconhecem a mesma linguagem é decidível. Seja

$$EQ_{\text{AFD}} = \{\langle A, B \rangle \mid A \text{ e } B \text{ são AFD's e } L(A) = L(B)\}.$$

Teorema 4.5

EQ_{AFD} é uma linguagem decidível.

Prova. Para provar esse teorema usamos o Teorema 4.4. Construimos um novo AFD C a partir de A e B , onde C aceita apenas aquelas cadeias que são aceitas por A ou B mas não por ambas. Por conseguinte, se A e B reconhecem a mesma linguagem C não aceitará nada. A linguagem de C é

$$L(C) = \left(L(A) \cap \overline{L(B)} \right) \cup \left(\overline{L(A)} \cap L(B) \right).$$

Essa expressão é às vezes chamada a **diferença simétrica** de $L(A)$ e $L(B)$ e é ilustrada na Figura 4.1. Aqui $\overline{L(A)}$ é o complemento de $L(A)$. A diferença simétrica é útil aqui porque $L(C) = \emptyset$ se e somente se $L(A) = L(B)$. Podemos construir C a partir de A e B com as construções para provar que a classe das linguagens regulares é fechada sob complementação, união, e interseção. Essas construções são algoritmos que podem ser levados adiante por máquinas de Turing. Uma vez que construimos C podemos usar o Teorema 4.4 para testar se $L(C)$ é vazia. Se ela é vazia, $L(A)$ e $L(B)$ têm que ser iguais.

F = “Sobre a entrada $\langle A, B \rangle$, onde A e B são AFD’s:

1. Construa o AFD C como descrito.
2. Rode a MT T do Teorema 4.4 sobre a entrada $\langle C \rangle$.
3. Se T aceita, *aceite*; caso contrário *rejeite*.

Figura 4.1: A diferença simétrica de $L(A)$ e $L(B)$ **Problemas decidíveis concernentes a linguagens livres-do-contesto**

Aqui, descrevemos algoritmos para testar se uma GLC gera uma cadeia específica e testar se a linguagem de uma GLC é vazia. Seja

$$A_{\text{GLC}} = \{\langle G, w \rangle \mid G \text{ é uma GLC que gera a cadeia } w\}.$$

Teorema 4.6
 A_{GLC} é uma linguagem decidível.

Idéia da prova. Para a GLC G e a cadeia w desejamos testar se G gera w . Uma idéia é usar G para passar por todas as derivações para determinar se alguma delas é uma derivação de w . Essa idéia não funciona, pois uma quantidade infinita de derivações pode ter que ser tentada. Se G não gera w , esse algoritmo nunca pararia. Essa idéia dá uma máquina de Turing que um reconhecedor, mas não um decisor, para A_{GLC} .

Para fazer dessa máquina de Turing um decisor precisamos de assegurar que o algoritmo tenta somente uma quantidade finita de derivações. No Problema 2.19 na página 121 mostramos que, se G estivesse na forma normal de Chomsky, qualquer derivação de w tem $2n - 1$ passos, onde n é o comprimento de w . Naquele caso verificar somente derivações com $2n - 1$ passos para determinar se G gera w seria suficiente. Somente uma quantidade finita de tais derivações existem. Podemos converter G para a forma normal de Chomsky usando o procedimento dado na Seção 2.1.

Prova. A MT S para A_{GLC} segue.

S = “Sobre a entrada $\langle G, w \rangle$, onde G é uma GLC e w é uma cadeia:

1. Converta G para uma gramática equivalente na forma normal de Chomsky.
2. Liste todas as derivações com $2n - 1$ passos, onde n é o comprimento de w , exceto se $n = 0$, então nesse caso liste todas as derivações com 1 passo.
3. Se qualquer dessas derivações gera w , aceite; se não, rejete.”

O problema de testar se uma GLC gera um cadeia específica está relacionado ao problema de compilar linguagens de programação. O algoritmo na MT S é muito ineficiente e nunca seria usado na prática, mas ele é fácil de descrever e não estamos preocupados com eficiência aqui. Na Parte Três deste livro lidamos com questões concernentes ao tempo de execução e o uso de memória de algoritmos. Na prova do Teorema 7.14, descrevemos um algoritmo mais eficiente para reconhecer linguagens livres-do-contesto.

Lembre-se que demos procedimentos para converter de um lado para o outro entre GLC's e AP's no Teorema 2.12. Portanto tudo que dizemos sobre a decidibilidade de problemas concernentes a GLC's aplica-se igualmente a AP's.

Vamos nos voltar agora para o problema de testar vacuidade para a linguagem de uma GLC. Como fizemos para AFD's, podemos mostrar que o problema de testar se uma GLC gera alguma cadeia é decidível. Seja

$$V_{\text{GLC}} = \{\langle G \rangle \mid G \text{ é uma GLC e } L(G) = \emptyset\}.$$

Teorema 4.7

V_{GLC} é uma linguagem decidível.

Idéia da prova. Para encontrar um algoritmo para esse problema podemos tentar usar a MT S do Teorema 4.6. Ele enuncia que podemos testar se uma GLC gera uma cadeia w específica. Para determinar se $L(G) = \emptyset$ o algoritmo poderia tentar passar por todas as possíveis w 's, uma por uma. Mas existe uma quantidade infinita de w 's para tentar, portanto esse método poderia terminar rodando para sempre. Precisamos de adotar uma abordagem diferente.

De modo a testar se a linguagem de uma gramática é vazia, precisamos testar se a variável inicial pode gerar uma cadeia de terminais. O algoritmo faz isso resolvendo um problema mais geral. Ele determina *para cada variável* se aquela variável é capaz de gerar uma cadeia de terminais. Quando o algoritmo determinou que uma variável pode gerar alguma cadeia de terminais, o algoritmo mantém registro dessa informação colocando uma marca sobre aquela variável.

Primeiro, o algoritmo marca todos os símbolos terminais na gramática. Então, ele varre todas as regras da gramática. Se ele por acaso encontra uma regra que permite alguma variável ser substituída por alguma cadeia de símbolos dos quais todos já estejam marcados, o algoritmo sabe que essa variável pode ser marcada também. O algoritmo continua dessa forma até que ele não possa marcar quaisquer variáveis adicionais. A MT R implementa esse algoritmo.

Prova.

R = “Sobre a entrada $\langle G \rangle$, onde G é uma GLC:

1. Marque todos os símbolos terminais de G .
2. Repita até que nenhuma variável nova venha a ser marcada.
3. Marque qualquer variável A onde G tem a regra $A \rightarrow U_1U_2 \cdots U_k$ e cada símbolo U_1, \dots, U_k já tenha sido marcado.
4. Se o símbolo inicial não está marcado, *aceite*; caso contrário *rejeite*.“

A seguir consideraremos o problema de testar se duas gramáticas livres-do-contesto geram a mesma linguagem. Seja

$$EQ_{\text{GLC}} = \{\langle G, H \rangle \mid G \text{ e } H \text{ são GLC's e } L(G) = L(H)\}.$$

O Teorema 4.5 deu um algoritmo que decide a linguagem análoga EQ_{AFD} para autômatos finitos. Usamos o procedimento de decisão para V_{AFD} para provar que EQ_{AFD} é decidível. Devido ao fato de que V_{GLC} também é decidível, você poderia pensar que podemos usar uma estratégia semelhante para provar que EQ_{GLC} é decidível. Mas algo dâ errado com essa idéia! A classe de linguagens livres-do-contesto *não* é fechada sob complementação ou interseção como você provou no Exercício 2.2. Na verdade, EQ_{GLC} não é decidível, e você verá a técnica para provar isso no Capítulo 5.

Agora mostramos que toda linguagem livre-do-contesto é decidível por uma máquina de Turing.

Teorema 4.8

Toda linguagem livre-do-contexto é decidível.

Idéia da prova. Seja A uma GLC. Nossa objetivo é mostrar que A é decidível. Uma (má) idéia é converter um AP para A diretamente numa MT. Isso não é difícil de fazer porque simular uma pilha com as fitas mais versáteis de MT's é fácil. O AP para A pode ser não-determinístico, mas isso parece legítimo porque você pode convertê-lo numa MT não-determinística e sabemos que qualquer MT não-determinística pode ser convertida numa MT determinística equivalente. Mesmo assim, há uma dificuldade. Alguns ramos da computação do AP pode seguir para sempre, lendo e escrevendo na pilha sem chegar numa parada. A MT simuladora então também teria alguns ramos não-terminantes na sua computação, e portanto a MT não seria um decisor. Uma idéia diferente é necessária. Ao invés, provamos esse teorema com a MT S que projetamos no Teorema 4.6 para decidir A_{GLC} .

Prova. Seja G uma GLC para A e projete uma MT M_G que decide A . Construimos uma cópia de G dentro de M_G . Ela funciona da seguinte maneira:

M = “Sobre a entrada w :

1. Rode a MT S sobre a entrada $\langle G, w \rangle$.
2. Se essa máquina aceita, *aceite*; se ela rejeita, *rejeite*.”

O Teorema 4.8 provê a ligação final no relacionamento entre as quatro principais classes de linguagens que descrevemos até agora neste curso: regulares, livres-do-contexto, decidíveis, e Turing-reconhecíveis. A Figura 4.2 ilustra esse relacionamento.

Figura 4.2: O relacionamento entre classes de linguagens

4.2 O problema da parada

Nesta seção provamos um dos teoremas mais filosoficamente importantes da teoria da computação: existe um problema específico que é algorítmicamente insolúvel. Computadores parecem ser tão poderosos que você pode acreditar que todos os problemas irão eventualmente se render a eles. O teorema apresentado aqui demonstra que computadores são limitados de uma maneira muito fundamental.

Que tipo de problemas são insolúveis por computador? Eles são esotéricos, residindo apenas nas mentes dos teóricos? Não! Até alguns problemas comuns que as pessoas desejam resolver acontecem de ser computacionalmente insolúveis.

Em um tipo de problemas insolúveis, você é apresentado a um programa de computador e uma especificação precisa do que aquele programa supostamente faz (e.g., ordenar uma lista de números). Devido ao fato de que ambos programa e especificação são objetos matematicamente precisos, você espera automatizar o processo de verificação alimentando esses objetos num computador programado apropriadamente. Entretanto,

você vai se desapontar. O problema geral de verificação de software não é solúvel por computador.

Nesta seção e no Capítulo 5 você vai encontrar problemas computacionalmente insolúveis. Nossos objetivos são ajudar você a desenvolver um sentimento para os tipos de problemas que são insolúveis e aprender técnicas para provar insolubilidade.

Agora nos voltamos para nosso primeiro teorema que estabelece a indecidibilidade de uma linguagem específica: o problema de testar se uma máquina de Turing aceita uma dada cadeia de entrada. Chamamo-lo A_{MT} por analogia com A_{AFD} e A_{GLC} . Mas, enquanto que A_{AFD} e A_{GLC} eram decidíveis, A_{MT} não o é. Seja

$$A_{MT} = \{\langle M, w \rangle \mid M \text{ é uma MT e } M \text{ aceita } w\}.$$

Teorema 4.9
 A_{MT} é indecidível.

Antes de chegar na prova, vamos primeiro observar que A_{MT} é Turing-reconhecível. Por conseguinte o Teorema 4.9 mostra que reconhecedores *são* mais poderosos que decisores. Exigir de uma MT que páre sobre todas as entradas restringe os tipos de linguagens que ela pode reconhecer. A seguinte máquina de Turing U reconhece A_{MT} .

U = “Sobre a entrada $\langle M, w \rangle$, onde M é uma MT e w é uma cadeia:

1. Simule M sobre a entrada w .
2. Se M alguma vez entra no seu estado de aceitação, *aceite*; se M alguma vez entra no seu estado de rejeição, *rejeite*.”

Note que essa máquina entra em loop sobre a entrada $\langle M, w \rangle$ se M entra em loop sobre w , e é por isso que essa máquina não decide A_{MT} . Se o algoritmo tivesse alguma maneira de determinar que M não estava parando sobre w , ele poderia *rejeitar*. Portanto A_{MT} é às vezes chamado o **problema da parada**. Como demonstramos, um algoritmo não tem forma de fazer essa determinação.

A máquina de Turing U é interessante em si própria. Ela é um exemplo da *máquina de Turing universal* originalmente proposta por Turing. Essa máquina é chamada universal porque ela é capaz de simular qualquer outra máquina de Turing a partir da descrição daquela máquina. A máquina de Turing universal desempenhou um importante papel inicial no estímulo ao desenvolvimento de computadores de programação armazenado.

O método da diagonalização

A prova da indecidibilidade do problema da parada usa uma técnica chamada *diagonalização*, descoberta pelo matemático Georg Cantor em 1873. Cantor estava preocupado com o problema de medir os tamanhos de conjuntos infinitos. Se temos dois conjuntos infinitos, como dizer se um é maior que o outro ou se eles são de mesmo tamanho? Para conjuntos finitos, é claro, responder a essas perguntas é fácil. Simplesmente contamos os elementos em um conjunto finito, e o número resultante é o seu tamanho. Mas, se tentarmos contar os elementos de um conjunto infinito, nunca terminaremos! Portanto não podemos usar o método de contagem para determinar os tamanhos relativos de conjuntos infinitos.

Por exemplo, tome o conjunto de inteiros pares e o conjunto de todos as cadeias sobre $\{0, 1\}$. Ambos os conjuntos são infinitos e por conseguinte maiores que qualquer

conjunto finito, mas um dos dois é maior que o outro? Como podemos comparar seu tamanho relativo?

Cantor propôs uma solução um tanto bela para esse problema. Ele observou que dois conjuntos finitos têm o mesmo tamanho se os elementos de um conjunto podem ser emparelhados com os elementos do outro conjunto. Esse método compara os tamanhos sem recorrer à contagem. Podemos estender essa idéia para conjuntos infinitos. Vamos ver o que ele significa mais precisamente.

Definição 4.10

Assuma que temos dois conjuntos A e B e uma função f de A para B . Digamos que f é **um-para-um** se ela nunca mapeia dois elementos diferentes ao mesmo lugar, isto é, se $f(a) \neq f(b)$ sempre que $a \neq b$. Digamos que f é **sobrejetora** se ela chegar em todo elemento de B , ou seja, se para todo $b \in B$ existe um $a \in A$ tal que $f(a) = b$. Digamos que A e B têm o **mesmo tamanho** se existe um função um-para-um, sobrejetora $f : A \rightarrow B$. Uma função que é tanto um-para-um quanto sobrejetora é chamada uma **correspondência**. Em uma correspondência todo elemento de A mapeia para um único elemento de B e cada elemento de B tem um único elemento de A mapeando para ele. Uma correspondência é simplesmente uma maneira de empregar os elementos de A com os elementos de B .

Exemplo 4.11

Seja \mathcal{N} o conjunto de números naturais $\{1, 2, 3, \dots\}$ e seja \mathcal{E} o conjunto dos números naturais pares $\{2, 4, 6, \dots\}$. Usando a definição de Cantor de tamanho podemos ver que \mathcal{N} e \mathcal{E} têm o mesmo tamanho. A correspondência f mapeando \mathcal{N} para \mathcal{E} é simplesmente $f(n) = 2n$. Podemos visualizar f mais facilmente com a ajuda de uma tabela.

n	$f(n)$
1	2
2	4
3	6
:	:

É claro que esse exemplo parece bizarro. Intuitivamente, \mathcal{E} é menor que \mathcal{N} porque \mathcal{E} é um subconjunto próprio de \mathcal{N} . Mas empregar cada membro de \mathcal{N} com seu próprio membro de \mathcal{E} é possível, portanto declararemos esses dois conjuntos como sendo de mesmo tamanho.

Definição 4.12

Um conjunto A é **contável** se ele é finito ou tem o mesmo tamanho que \mathcal{N} .

Exemplo 4.13

Agora nos voltamos para um exemplo ainda mais estranho. Se fizermos \mathcal{Q} ser o conjunto dos números racionais positivos, ou seja, $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$, \mathcal{Q} parece ser muito maior que \mathcal{N} . Mesmo assim esses dois conjuntos são do mesmo tamanho. Demonstramos essa conclusão dando uma correspondência com \mathcal{N} para mostrar que \mathcal{Q} é contável. Uma maneira fácil de dar uma correspondência com \mathcal{N} é listar todos os elementos de \mathcal{Q} . Então empregamos o primeiro elemento na lista com o número 1 de

\mathcal{N} , o segundo elemento na lista com o número 2 de \mathcal{N} , e assim por diante. Temos que verificar para ter certeza de que todo membro de \mathcal{Q} aparece apenas uma vez na lista.

Para obter essa lista fazemos uma matriz infinita contendo todos os membros dos racionais positivos, como mostrado na Figura 4.3. A i -ésima linha contém todos os números com numerador i e a j -ésima coluna tem todos os números com denominador j . Portanto o número $\frac{i}{j}$ ocorre na i -ésima linha e j -ésima coluna.

Agora transformamos essa matriz numa lista. Uma (má) forma de tentá-lo seria começar a lista com todos os elementos na primeira linha. Essa não é uma boa abordagem porque a primeira linha é infinita, portanto a lista nunca chegaria na segunda linha. Ao invés disso, listamos os elementos nas diagonais, começando do canto, que estão superimpostas no diagrama. A primeira diagonal contém um único elemento $\frac{1}{1}$, e a segunda diagonal contém os dois elementos $\frac{2}{1}$ e $\frac{1}{2}$. Portanto os primeiros três elementos na lista são $\frac{1}{1}$, $\frac{2}{1}$, e $\frac{1}{2}$. Na terceira diagonal uma complicação aparece. Ela conteém $\frac{3}{1}$, $\frac{2}{2}$, e $\frac{1}{3}$. Se simplesmente adicionarmos esses à lista, repeteríamos $\frac{1}{1} = \frac{2}{2}$. Evitamos fazer isso pulando um elemento quando ele causaria uma repetição. Portanto adicionamos somente os dois novos elementos $\frac{3}{1}$ e $\frac{1}{3}$. Continuando dessa maneira obtemos uma lista de todos os elementos de \mathcal{Q} .

Figura 4.3: Uma correspondência de \mathcal{N} e \mathcal{Q}

Após ver a correspondência de \mathcal{N} e \mathcal{Q} , você poderia pensar que quaisquer dois conjuntos infinitos podem ser mostrados ter o mesmo tamanho. Afinal de contas, você precisa somente demonstrar uma correspondência, e esse exemplo mostra que correspondências surpreendentes realmente existem. Entretanto, para alguns conjuntos infinitos nenhuma correspondência com \mathcal{N} existe. Esses conjuntos são simplesmente grandes demais. Tais conjuntos são chamados *incontáveis*.

O conjunto dos números reais é um exemplo de um conjunto incontável. Um **número real** é aquele que tem uma representação decimal. Os números $\pi = 3,14159265\dots$ e $\sqrt{2} = 1,4142135\dots$ são exemplos de números reais. Seja \mathcal{R} o conjunto de números reais. Cantor provou que \mathcal{R} é incontável. Fazendo isso ele introduziu o método da diagonalização.

Teorema 4.14
 \mathcal{R} é incontável.

Prova. Para mostrar que \mathcal{R} é incontável, mostramos que nenhuma correspondência existe entre \mathcal{N} e \mathcal{R} . A prova é por contradição. Suponha que uma correspondência f existisse entre \mathcal{N} e \mathcal{R} . Nossa tarefa é mostrar que f falha em funcionar como deveria. Para ela ser uma correspondência, f tem que emparelhar todos os membros de \mathcal{N} com todos os membros de \mathcal{R} . Mas encontraremos um x em \mathcal{R} que não está emparelhado com nada em \mathcal{N} , o que será nossa contradição.

A maneira pela qual encontramos esse x é verdadeiramente construindo-o. Escolhemos cada dígito de x para fazer x diferente de um dos números reais que está emparelhado com um elemento de \mathcal{N} . No final temos certeza de que x é diferente de qualquer número real que está emparelhado.

Podemos ilustrar essa idéia dando um exemplo. Suponha que a correspondência f existe. Faça $f(1) = 3,14159\dots$, $f(2) = 55,55555\dots$, $f(3) = \dots$, e assim por diante,

só para mencionar alguns valores de f . Então f emparelha o número 1 com 3, 14159..., o número 2 com 55, 55555..., e assim por diante. A seguinte tabela mostra uns poucos valores de uma correspondência hipotética f entre \mathcal{N} e \mathcal{R} .

n	$f(n)$
1	3, 14159...
2	55, 55555...
3	0, 12345...
4	0, 50000...
:	:

Construimos o x desejado dando sua representação decimal. É um número entre 0 e 1, de modo que todos os dígitos significativos são dígitos fracionários seguintes à vírgula decimal. Nossa objetivo é assegurar que $x \neq f(n)$ para qualquer n . Para assegurar que $x \neq f(1)$ fazemos com que o primeiro dígito de x seja qualquer coisa diferente do primeiro dígito fracionário 1 de $f(1) = 3, 14159\dots$. Arbitrariamente, fazemos com que ele seja 4. Para assegurar que $x \neq f(2)$ fazemos o com que o segundo dígito de x seja qualquer coisa diferente do segundo dígito fracionário 5 de $f(2) = 55, 55555\dots$. Arbitrariamente, fazemos com que ele seja 6. O terceiro dígito fracionário de $f(3) = 0, 12345\dots$ é 3, portanto fazemos com que x seja qualquer coisa diferente, digamos, 4. Continuando dessa maneira descendo na diagonal da tabela para f , obtemos todos os dígitos de x , como mostrado na tabela abaixo. Sabemos que x não é $f(n)$ para qualquer que seja n porque ele difere de $f(n)$ no n -ésimo dígito fracionário. (Um pequeno problema surge porque certos números, tais como 0, 1999... e 0, 2000..., são iguais muito embora suas representações decimais sejam diferentes. Evitamos esse problema nunca selecionando os dígitos 0 ou 9 quando construimos x .)

n	$f(n)$
1	3, <u>1</u> 4159...
2	55, <u>5</u> 5555...
3	0, <u>1</u> 2345...
4	0, 5 <u>0</u> 000...
:	:

$x = 0, 4641\dots$

O teorema precedente tem uma importante aplicação à teoria da computação. Ele mostra que algumas linguagens não são decidíveis ou mesmo Turing-reconhecíveis, pela razão que existe uma quantidade incontável de linguagens e mesmo assim somente uma quantidade contável de máquinas de Turing. Devido ao fato de que cada máquina de Turing pode reconhecer uma única linguagem e que existem mais linguagens que máquinas de Turing, algumas linguagens não são reconhecíveis por nenhuma máquina de Turing. Tais linguagens não são Turing-reconhecíveis, como enunciamos no corolário a seguir.

Corolário 4.15

Algumas linguagens não são Turing-reconhecíveis.

Prova. Para mostrar que o conjunto de todas as máquinas de Turing é contável primeiro observamos que o conjunto de todas as cadeias Σ^* é contável, para qualquer alfabeto Σ . Com apenas uma quantidade finita de cadeias de cada comprimento, podemos

formar uma lista de Σ^* relacionando todas as cadeias de comprimento 0, comprimento 1, comprimento 2, e assim por diante.

O conjunto de todas as máquinas de Turing é contável porque cada máquina de Turing M tem uma codificação numa cadeia $\langle M \rangle$. Se simplesmente omitirmos aquelas cadeias que não são codificações legítimas de máquinas de Turing, podemos obter uma lista de todas as máquinas de Turing.

Para mostrar que o conjunto de todas as linguagens é incontável primeiro observamos que o conjunto de todas as seqüências binárias infinitas é incontável. Uma *seqüência binária infinita* é uma seqüência sem-fim de 0's e 1's. Suponha que \mathcal{B} seja o conjunto de todas as seqüências binárias infinitas. Podemos mostrar que \mathcal{B} é incontável usando uma prova por diagonalização semelhante àquela usada na Teorema 4.14 para mostrar que \mathcal{R} é incontável.

Seja \mathcal{L} o conjunto de todas as linguagens sobre o alfabeto Σ . Mostramos que \mathcal{L} é incontável dando uma correspondência com \mathcal{B} , portanto mostrando que os dois conjuntos são do mesmo tamanho. Seja $\Sigma^* = \{s_1, s_2, s_3, \dots\}$. Cada linguagem $A \in \mathcal{L}$ tem uma única seqüência em \mathcal{B} . O i -ésimo bit daquela seqüência é um 1 se $s_i \in A$ e é um 0 se $s_i \notin A$, o que é chamado a *seqüência característica* de A . Por exemplo, se A fosse a linguagem de todas as cadeias começando com um 0 sobre o alfabeto $\{0, 1\}$, sua seqüência característica χ_A seria

$$\begin{aligned} \Sigma^* &= \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots \}; \\ A &= \{ , 0, , 00, 01, , , 000, 001, \dots \}; \\ \chi_A &= \begin{matrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \dots \end{matrix}. \end{aligned}$$

A função $f : \mathcal{L} \rightarrow \mathcal{B}$, onde $f(A)$ é igual à seqüência característica de A , é injetora e sobrejetora e portanto uma correspondência. Por conseguinte, como \mathcal{B} é incontável, \mathcal{L} também é incontável.

Portanto mostramos que o conjunto de todas as linguagens não pode ser posto numa correspondência com o conjunto de todas as máquinas de Turing. Concluimos que algumas linguagens não são reconhecidas por nenhuma máquina de Turing.

O problema da parada é indecidível

Agora estamos prontos para provar o Teorema 4.9, a indecidibilidade da linguagem

$$A_{\text{MT}} = \{\langle M, w \rangle \mid M \text{ é uma MT e } M \text{ aceita } w\}.$$

Prova. Assumimos que A_{MT} é decidível e obtemos uma contradição. Suponha que H é um decisor para A_{MT} . Sobre a entrada $\langle M, w \rangle$, onde M é uma MT e w é uma cadeia, H pára e aceita se M aceita w . Além disso, H pára e rejeita se M falha em aceita w . Em outras palavras, assumimos que H é uma MT, onde

$$H(\langle M, w \rangle) = \begin{cases} \text{aceita} & \text{se } M \text{ aceita } w \\ \text{rejeita} & \text{se } M \text{ não aceita } w. \end{cases}$$

Agora construímos uma nova máquina de Turing D com H como uma subrotina. Essa nova MT chama H para determinar o que M faz quando a entrada para M é sua própria descrição $\langle M \rangle$. Uma vez que D tenha determinado essa informação, ela faz o oposto. Ou seja, ela rejeita se M aceita e aceita se M não aceita. Abaixo está uma descrição de D :

D = “Sobre a entrada $\langle M \rangle$, onde M é uma MT

1. Rode H sobre a entrada $\langle M, \langle M \rangle \rangle$.
2. Dê como saída o oposto do que H dá como saída, ou seja, se H aceita, *rejeite* e se H rejeita, *aceite*."

Não se confunda com a idéia de rodar uma máquina sobre sua própria descrição! Isso é semelhante a rodar um programa com ele próprio como entrada, algo que realmente ocorre eventualmente na prática. Por exemplo, um compilador é um programa que traduz outros programas. Um compilador para a linguagem Pascal pode ele próprio ser escrito em Pascal, de modo que rodar aquele programa sobre si próprio faria sentido. Em resumo,

$$D(\langle M \rangle) = \begin{cases} \text{aceita} & \text{se } M \text{ não aceita } \langle M \rangle \\ \text{rejeita} & \text{se } M \text{ aceita } \langle M \rangle. \end{cases}$$

O que acontece quando rodamos D com sua própria descrição $\langle D \rangle$ como entrada? Nesse caso obtemos

$$D(\langle D \rangle) = \begin{cases} \text{aceita} & \text{se } D \text{ não aceita } \langle D \rangle \\ \text{rejeita} & \text{se } D \text{ aceita } \langle D \rangle. \end{cases}$$

Independentemente do que D faz, ela é forçada a fazer o oposto, o que é obviamente uma contradição. Por conseguinte nem a MT D nem a MT M podem existir.

Vamos revisar os passos dessa prova. Assuma que uma MT H decide A_{MT} . Então use H para construir uma MT D que quando recebe a entrada $\langle M \rangle$ aceita exatamente quando M não aceita a entrada $\langle M \rangle$. Finalmente, rode D sobre si própria. As máquinas toma as seguintes ações, com a última linha sendo a contradição.

- H aceita $\langle M, w \rangle$ exatamente quando M aceita w .
- D rejeita $\langle M \rangle$ exatamente quando M aceita $\langle M \rangle$.
- D rejeita $\langle D \rangle$ exatamente quando D aceita $\langle D \rangle$.

Onde está a diagonalização na prova do Teorema 4.9? Ele se torna aparente quando você examina as tabelas de comportamento para as MT's H e D . Nessas tabelas listamos todas as MT's nas linhas, M_1, M_2, \dots e todas as suas descrições nas colunas, $\langle M_1 \rangle, \langle M_2 \rangle, \dots$ As entradas dizem se a máquina em uma dada linha aceita a entrada em uma dada coluna. A entrada é *aceita* se a máquina aceita a entrada mas está em branco se ela rejeita ou entra em loop sobre aquela entrada. Montamos as entradas na figura abaixo para ilustrar a idéia.

Na figura seguinte as entradas são os resultados de rodar H sobre as entradas correspondentes na Figura 4.4. Portanto se M_3 não aceita a entrada $\langle M_2 \rangle$, a entrada para linha M_3 e coluna $\langle M_2 \rangle$ é *rejeita* porque H rejeita a entrada $\langle M_3, \langle M_2 \rangle \rangle$.

Na figura a seguir adicionamos D à Figura 4.5. Pela nossa hipótese, H é uma MT e portanto D também o é. Por conseguinte ela tem que ocorrer na lista M_1, M_2, \dots de todas as MT's. Note que D computa o oposto das entradas da diagonal. A contradição ocorre no ponto onde está a interrogação pois a entrada aí tem que ser o oposto de si próprio.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots
M_1	aceita		aceita		
M_2	aceita	aceita	aceita	aceita	
M_3					\dots
M_4	aceita	aceita			
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots

Figura 4.4: Entrada i, j é aceita se M_i aceita $\langle M_j \rangle$

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots
M_1	aceita	rejeita	aceita	rejeita	
M_2	aceita	aceita	aceita	aceita	
M_3	rejeita	rejeita	rejeita	rejeita	\dots
M_4	aceita	aceita	rejeita	rejeita	
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots

Figura 4.5: A entrada i, j é o valor de H sobre a entrada $\langle M_i, \langle M_j \rangle \rangle$

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots	$\langle D \rangle$	\dots
M_1	aceita	rejeita	aceita	rejeita		aceita	
M_2	aceita	aceita	aceita	aceita		aceita	
M_3	rejeita	rejeita	rejeita	rejeita	\dots	rejeita	\dots
M_4	aceita	aceita	rejeita	rejeita		aceita	
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
D	rejeita	rejeita	aceita	aceita		?	
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots

Figura 4.6: Se D está na figura, uma contradição ocorre em “?”

Uma linguagem Turing-irreconhecível

Na seção precedente exibimos uma linguagem, a saber, A_{MT} , que é indecidível. Agora exibimos uma linguagem que não é sequer Turing-reconhecível. Note que A_{MT} não bastará para esse propósito porque mostramos que A_{MT} é Turing-reconhecível na página 160. O teorema a seguir mostra que, se tanto uma linguagem quanto seu complemento são Turing-reconhecíveis, a linguagem é decidível. Portanto, para qualquer linguagem indecidível, ou ela ou seu complemento não é Turing-reconhecível. Lembre-se que o complemento de uma linguagem é a linguagem consistindo de todas as cadeias que não estão na linguagem. Dizemos que uma linguagem é *co-Turing-reconhecível* se ela é o complemento de uma linguagem Turing-reconhecível.

Teorema 4.16

Uma linguagem é decidível se e somente se ela é tanto Turing-reconhecível quanto co-Turing-reconhecível.

Em outras palavras, uma linguagem é decidível se e somente se tanto ela quanto seu complemento são Turing-reconhecíveis.

Prova. Temos duas direções para provar. Primeiro, se A é decidível, podemos facilmente ver que tanto A quanto seu complemento \bar{A} são Turing-reconhecíveis. Qualquer linguagem decidível é Turing-reconhecível, e o complemento de uma linguagem decidível também é decidível.

Para a outra direção, se tanto A quanto \bar{A} são Turing-reconhecíveis, tomamos M_1 como sendo o reconhecedor para A e M_2 como o reconhecedor para \bar{A} . A seguinte máquina de Turing M é um decisor para A .

M = “Sobre a entrada w :

1. Rode ambas M_1 e M_2 sobre a entrada w em paralelo.
2. Se M_1 aceita, *aceite*; se M_2 aceita, *rejeite*.”

Rodar as duas máquinas em paralelo significa que M tem duas fitas, uma para simular M_1 e outra para simular M_2 . Nesse caso M alterna simulando um passo de cada máquina, o que continua até que uma delas páre.

Agora mostramos que M decide A . Toda cadeia w está em A ou em \bar{A} . Por conseguinte ou M_1 ou M_2 tem que aceitar w . Devido ao fato de que M pára sempre que M_1 ou M_2 aceita, M sempre pára e portanto ela é um decisor. Além do mais, ela aceita todas as cadeias em A e rejeita todas as cadeias que não estão em A . Portanto M é um decisor para A , e por conseguinte A é decidível.

Corolário 4.17

$\overline{A_{MT}}$ não é Turing-reconhecível.

Prova. Sabemos que A_{MT} é Turing-reconhecível. Se $\overline{A_{MT}}$ também fosse Turing-reconhecível, A_{MT} seria decidível. O Teorema 4.9 nos diz que A_{MT} não é decidível, portanto $\overline{A_{MT}}$ tem que não-Turing-reconhecível.

.....

Exercícios

- 4.1** Responda todos os itens para o AFD M da Figura 4.7 e dê razões para sua resposta

Figura 4.7:

- a. $\langle M, 0100 \rangle \in A_{\text{AFD}}$?
- b. $\langle M, 011 \rangle \in A_{\text{AFD}}$?
- c. $\langle M \rangle \in A_{\text{AFD}}$?
- d. $\langle M, 0100 \rangle \in A_{\text{EXR}}$?
- e. $\langle M \rangle \in V_{\text{AFD}}$?
- f. $\langle M, M \rangle \in EQ_{\text{AFD}}$?

- 4.2** Considere o problema de se testar se um AFD e uma expressão regular são equivalentes. Expresse esse problema como uma linguagem e mostre que ela é decidível.

- 4.3** Seja $TOD_{\text{AFD}} = \{\langle A \rangle \mid A \text{ é um AFD que reconhece } \Sigma^*\}$. Mostre que TOD_{AFD} é decidível.

- 4.4** $A\varepsilon_{\text{GLC}} = \{\langle G \rangle \mid G \text{ é uma GLC que gera } \varepsilon\}$. Mostre que $A\varepsilon_{\text{GLC}}$ é decidível.

- 4.5** Seja $INFINITA_{\text{AFD}} = \{\langle A \rangle \mid A \text{ é um AFD e } L(A) \text{ é uma linguagem infinita}\}$. Mostre que $INFINITA_{\text{AFD}}$ é decidível.

- 4.6** Seja X o conjunto $\{1, 2, 3, 4, 5\}$ e Y o conjunto $\{6, 7, 8, 9, 10\}$. Descrevemos as funções $f : X \rightarrow Y$ e $g : X \rightarrow Y$ nas seguintes tabelas:

n	$f(n)$	n	$g(n)$
1	6	1	10
2	7	2	9
3	6	3	8
4	7	4	7
5	6	5	6

- a. f é injetora? g é injetora? Se não, diga por que.
- b. f é sobrejetora? g é sobrejetora? Se não, diga por que.
- c. f é uma correspondência? g é uma correspondência? Se não, diga por que.

- 4.7** Seja \mathcal{B} o conjunto de todas as seqüências infinitas sobre $\{0, 1\}$. Mostre que \mathcal{B} é incontável, usando uma prova por diagonalização.

- 4.8** Seja $T = \{(i, j, k) \mid i, j, k \in \mathbb{N}\}$. Mostre que T é contável.

- 4.9** Revise a forma como definimos conjuntos como tendo o mesmo tamanho na Definição 4.10 na página 161. Mostre que “é do mesmo tamanho que” é uma relação de equivalência.

Problemas

4.10 Seja

$$A = \{\langle M \rangle \mid M \text{ é um AFD que não aceita nenhuma cadeia contendo um número ímpar de } 1\text{'s}\}.$$

Mostre que A é decidível.

4.11 Seja $A = \{\langle R, S \rangle \mid R \text{ e } S \text{ são expressões regulares e } L(R) \subseteq L(S)\}$. Mostre que A é decidível.

4.12 Mostre que o problema de se testar se uma GLC gera alguma cadeia em 1^* é decidível. Em outras palavras, mostre que $\{\langle G \rangle \mid G \text{ é uma GLC sobre } \{0, 1\}^* \text{ e } 1^* \cap L(G) \neq \emptyset\}$ é uma linguagem decidível.

***4.13** Mostre que o problema de se testar se uma GLC gera todas as cadeias em 1^* é decidível. Em outras palavras, mostre que $\{\langle G \rangle \mid G \text{ é uma GLC sobre } \{0, 1\}^* \text{ e } 1^* \subseteq L(G)\}$ é uma linguagem decidível.

4.14 Seja $A = \{\langle R \rangle \mid R \text{ é uma expressão regular descrevendo uma linguagem contendo pelo menos uma cadeia } w \text{ que tem } 111 \text{ como uma subcadeia (i.e., } w = x111y \text{ para algum } x \text{ e algum } y\}\}$. Mostre que A é decidível.

***4.15** Seja $E = \{\langle M \rangle \mid M \text{ é um AFD que aceita alguma cadeia da forma } ww^R \text{ para } w \in \{0, 1\}^a st\}$.

Mostre que E é decidível.

4.16 Prove que EQ_{AFD} é decidível testando os dois AFD's sobre todas as cadeias até um certo tamanho. Calcule um tamanho que funcione.

4.17 Seja C uma linguagem. Prove que C é Turing-reconhecível se e só se existir uma linguagem decidível D tal que $C = \{x \mid \exists y (\langle x, y \rangle \in D)\}$.

4.18 Sejam A e B duas linguagens disjuntas. Digamos que a linguagem C *separa* A e B se $A \subseteq C$ e $B \subseteq \bar{C}$. Mostre que quaisquer duas linguagens co-Turing-reconhecíveis são separáveis por alguma linguagem decidível.

4.19 Seja $S = \{\langle M \rangle \mid M \text{ é um AFD que aceita } w^R \text{ sempre que ele aceita } w\}$. Mostre que S é decidível.

4.20 Um *estado inútil* em um autômato a pilha nunca é atingido sobre qualquer que seja a entrada. Considere o problema de testar se se um autômato a pilha tem quaisquer estados inúteis. Formule esse problema como uma linguagem e mostre que ele é decidível.

4.21 Seja A uma linguagem Turing-reconhecível consistindo de descrições de máquinas de Turing, $\{\langle M_1 \rangle, \langle M_2 \rangle, \dots\}$, onde toda M_i é um decisor. Prove que alguma linguagem decidível D não é decidida por nenhum decisor M_i cuja descrição aparece em A . (Dica: Você pode achar útil considerar um enumerador para A .)

***4.22** Seja B uma linguagem Turing-reconhecível consistindo de descrições de máquinas de Turing, $\{\langle M_1 \rangle, \langle M_2 \rangle, \dots\}$. Mostre que existe uma linguagem decidível C consistindo de máquinas de Turing tal que toda máquina descrita em B tem uma equivalente em C e vice-versa.

Capítulo 5

Redutibilidade

No Capítulo 4 estabelecemos a máquina de Turing como nosso modelo de um computador de propósito geral. Apresentamos vários exemplos de problemas que são solúveis sobre uma máquina de Turing e demos um exemplo de um problema, A_{MT} , que é computacionalmente insolúvel. Neste capítulo examinamos vários problemas insolúveis adicionais. Ao fazer isso introduzimos o método principal para provar que problemas são computacionalmente insolúveis. Ele é chamado **redutibilidade**.

Uma **redução** é uma maneira de converter um problema em um outro problema de tal maneira que uma solução para o segundo problema pode ser usada para resolver o primeiro problema. Tais redutibilidades surgem frequentemente no cotidiano, mesmo se não nos refiramos a eles usualmente dessa maneira. Por exemplo, suponha que você queira saber andar em uma nova cidade. Você sabe que isso seria fácil se você tivesse um mapa. Por conseguinte você pode reduzir o problema de saber andar na cidade ao problema de obter um mapa da cidade.

Redutibilidade sempre envolve dois problemas, que chamamos A e B . Se A se reduz para B , podemos usar uma solução para B para resolver A . Portanto em nosso exemplo, A é o problema de saber andar na cidade e B é o problema de obter um mapa. Note que redutibilidade não diz nada sobre resolver A ou B sozinhos, mas somente a solubilidade de A na presença de uma solução para B .

A seguir estão mais exemplos de redutibilidades. O problema de viajar de Boston a Paris se reduz ao problema de comprar uma passagem aérea entre as duas cidades. Esse problema por sua vez se reduz ao problema de ganhar dinheiro para a passagem. E esse problema se reduz ao problema de encontrar um emprego. Redutibilidade também ocorre em problemas matemáticos. Por exemplo, o problema de medir a área de um retângulo se reduz ao problema de medir sua altura e sua largura. O problema de resolver um sistema de equações lineares se reduz ao problema de inverter uma matriz.

Redutibilidade desempenha um papel importante na classificação de problemas por decidibilidade e mais tarde em teoria da complexidade também. Quando A é redutível a B , resolver A não pode ser mais difícil que resolver B porque uma solução para B dá uma solução para A . Em termos de teoria da computabilidade, se A é redutível a B e B é decidível, A também é decidível. Equivalentemente, se A é indecidível e redutível a B , B é indecidível. Essa última versão é a chave para se provar que vários problemas são indecidíveis.

Em resumo, nosso método para provar que um problema é indecidível será: Mostre que algum problema já conhecido como indecidível se reduz a ele.

5.1 Problemas indecidíveis da teoria de linguagens....

Já estabelecemos a indecidibilidade de A_{MT} , o problema de se determinar se uma máquina de Turing aceita uma dada entrada. Vamos considerar um problema relacionado, $PARA_{MT}$, o problema de se determinar se uma máquina de Turing pára (aceitando ou rejeitando) sobre uma dada entrada.¹ Usamos a indecidibilidade de A_{MT} para provar a indecidibilidade de $PARA_{MT}$ reduzindo A_{MT} a $PARA_{MT}$. Seja

$$PARA_{MT} = \{\langle M, w \rangle \mid M \text{ é uma MT e } M \text{ pára sobre a entrada } w\}.$$

Teorema 5.1
 $PARA_{MT}$ é indecidível.

.....
Idéia da Prova. Esta prova é por contradição. Assumimos que $PARA_{MT}$ é decidível e usamos a suposição para mostrar que A_{MT} é decidível, contradizendo o Teorema 4.9. A idéia chave é mostrar que A_{MT} é redutível a $PARA_{MT}$.

Vamos assumir que temos uma MT R que decide $PARA_{MT}$. Então usamos R para construir S , uma MT que decide A_{MT} . Para ter uma idéia de como construir S , faça de conta que você é S . Sua tarefa é decidir A_{MT} . Você recebe uma entrada da forma $\langle M, w \rangle$. Você tem que dar como saída *aceita* se M aceita w , e você tem que dar como saída *rejeita* se M entra em loop ou rejeita sobre w . Tente simular M sobre w . Se ela aceita ou rejeita, faça o mesmo. Mas você pode não ser capaz de determinar se M está em loop, e nesse caso sua simulação não terminará. Isso é ruim, pois você é um decisor e portanto nunca lhe é permitido entrar em loop. Portanto essa idéia, em si, não funciona.

Ao invés disso, use a suposição de que você tem uma MT R que decide $PARA_{MT}$. Com R , você pode testar se M pára sobre w . Se R indica que M não pára sobre w , rejeite porque $\langle M, w \rangle$ não está em A_{MT} . Entretanto, se R indica que M de fato pára sobre w , você pode fazer a simulação sem qualquer perigo de entrar em loop.

Por conseguinte, se a MT R existe, você pode decidir A_{MT} , mas sabemos que A_{MT} é indecidível. Em virtude dessa contradição podemos concluir que R não pode existir. Portanto $PARA_{MT}$ é indecidível.

Prova. Vamos assumir com o propósito de obter uma contradição que a MT R decide $PARA_{MT}$. Construimos a MT S para decidir A_{MT} , com S operando da seguinte forma. $S =$ “Sobre a entrada $\langle M, w \rangle$, uma codificação de uma MT M e uma entrada w :

1. Rode a MT R sobre a entrada $\langle M, w \rangle$.
2. Se R rejeita, *rejeite*.
3. Se R aceita, simule M sobre w até que ela páre.
4. Se M aceitou, *aceite*; se M rejeitou, *rejeite*.”

¹Na Seção 4.2, usamos o termo *problema da parada* para a linguagem A_{MT} muito embora $PARA_{MT}$ seja o real problema da parada. Daqui por diante distinguimos os dois chamando A_{MT} de o *problema da aceitação*.

Claramente, se R decide $PARA_{MT}$, então S decide A_{MT} . Como A_{MT} é indecidível, $PARA_{MT}$ também tem que ser indecidível.

O Teorema 5.1 ilustra nossa estratégia para provar que um problema é indecidível. Essa estratégia é comum à maioria das provas de indecidibilidade, exceto para a indecidibilidade de A_{MT} propriamente dita, que é provada diretamente via o método da diagonalização.

Agora apresentamos vários outros teoremas e suas provas como mais exemplos do método da reducibilidade para provar a indecidibilidade. Seja

$$V_{MT} = \{\langle M, \rangle \mid M \text{ é uma MT e } L(M) = \emptyset\}.$$

Teorema 5.2
 V_{MT} é indecidível.

Idéia da prova. Seguimos o padrão adotado no Teorema 5.1. Assumimos para os propósitos de obter uma contradição que V_{MT} é decidível e então mostramos que A_{MT} é decidível—uma contradição. Seja R uma MT que decide V_{MT} . Usamos R para construir uma MT S que decide A_{MT} . Como funcionará S quando ela recebe a entrada $\langle M, w \rangle$?

Uma idéia é para S rodar R sobre a entrada $\langle M \rangle$ e ver se ela aceita. Se sim, sabemos que $L(M)$ é vazia e portanto que M não aceita w . Mas, se R rejeita $\langle M \rangle$, tudo o que sabemos é que $L(M)$ não é vazia e por conseguinte que M aceita alguma cadeia, mas ainda não sabemos se M aceita a cadeia específica w . Portanto precisamos usar uma idéia diferente.

Ao invés de rodar R sobre $\langle M \rangle$ rodamos R sobre uma modificação de $\langle M \rangle$. Modificamos $\langle M \rangle$ para garantir que M rejeita todas as cadeias exceto w , mas sobre a entrada w ela funciona como de costume. Então usamos R para testar se a máquina modificada reconhece a linguagem vazia. A única cadeia que a máquina pode agora aceitar é w , portanto sua linguagem será não-vazia se e somente se ela aceita w . Se R aceita quando ela é alimentada com uma descrição da máquina modificada, sabemos que a máquina modificada não aceita nada e que M não aceita w .

Prova. Vamos escrever a máquina modificada descrita na idéia da prova usando nossa notação padrão. Chamamo-la M_1 .

M_1 = “Sobre a entrada x :

1. Se $x \neq w$, *rejeite*.
2. Se $x = w$, rode M sobre a entrada w e *aceite* se M aceita.”

Essa máquina tem a cadeia w como parte de sua descrição. Ela conduz o teste de se $x = w$ da maneira óbvia, fazendo uma varredura na entrada e comparando-a caracter a caracter com w para determinar se eles são o mesmo.

Colocando tudo isso junto, assumimos que a MT R decide V_{MT} e construímos a MT S que decide A_{MT} da seguinte maneira.

S = “Sobre a entrada $\langle M, w \rangle$, uma codificação de uma MT M e uma cadeia w :

1. Use a descrição de M e w e construa a MT M_1 descrita acima.

2. Rode R sobre a entrada $\langle M_1 \rangle$.
3. Se R aceita, *rejeite*; se R rejeita, *aceite*."

Note que S tem na realidade que ser capaz de computar uma descrição de M_1 a partir de uma descrição de M e w . Ela é capaz de fazer isso porque ela precisa apenas adicionar estados extra a M que realizem o teste $x = w$.

Se R fosse um decisor para V_{MT} , S seria um decisor para A_{MT} . Um decisor para A_{MT} não pode existir, portanto sabemos que V_{MT} tem que ser indecidível.

.....

Um outro problema computacional interessante acerca de máquinas de Turing concerne testar se uma dada máquina de Turing reconhece uma linguagem que também pode ser reconhecida por um modelo computacional mais simples. Por exemplo suponhamos que $\text{REGULAR}_{\text{MT}}$ seja o problema de se testar se uma dada máquina de Turing tem um autômato finito que lhe é equivalente. Esse problema é o mesmo que testar se a máquina de Turing reconhece uma linguagem regular. Seja

$$\text{REGULAR}_{\text{MT}} = \{\langle M \rangle \mid M \text{ é uma MT e } L(M) \text{ é uma linguagem regular}\}.$$

Teorema 5.3

$\text{REGULAR}_{\text{MT}}$ é indecidível.

.....

Idéia da prova. Como de costume para teoremas de indecidibilidade, esta prova é por redução a partir de A_{MT} . Assumimos que $\text{REGULAR}_{\text{MT}}$ é decidível por uma MT R e usamos essa suposição para construir uma MT S que decide A_{MT} . Menos óbvio agora é como usar a capacidade de R ajudar S nessa tarefa. Não obstante, podemos fazê-lo.

A idéia é que S tome sua entrada $\langle M, w \rangle$ e modifique M de tal modo que a MT resultante reconheça uma linguagem regular se e somente se M aceita w . Chamamos a máquina modificada M_2 . Projetamos M_2 para reconhecer a linguagem não-regular $\{0^n 1^n \mid n \geq 0\}$ se M não aceita w e a linguagem regular Σ^* se M aceita w . Temos que especificar como S pode construir tal M_2 a partir de M e w . Aqui, M_2 funciona automaticamente aceitando todas as cadeias em $\{0^n 1^n \mid n \geq 0\}$. Adicionalmente, se M aceita w , M_2 aceita todas as outras cadeias.

Prova. Supomos que R seja uma MT que decide $\text{REGULAR}_{\text{MT}}$ e construimos uma MT S para decidir A_{MT} . Então S funciona da seguinte maneira.

$S =$ "Sobre a entrada $\langle M, w \rangle$ onde M é uma MT e w é uma cadeia:

1. Construaa seguinte MT M_2 .

$M_2 =$ "Sobre a entrada x :

1. Se x tem a forma $0^n 1^n$, *aceite*.
2. Se x não tem essa forma, rode M sobre a entrada w e *aceite* se M aceita w .
2. Rode R sobre a entrada $\langle M_2 \rangle$.
3. Se R aceita, *aceite*; se R rejeita, *rejeite*.

.....

Similarmente, os problemas de se testar se a linguagem de uma máquina de Turing é uma linguagem livre-do-contesto, uma linguagem decidível, ou mesmo uma linguagem finita, podem ser mostrados serem indecidíveis com provas semelhantes. Na verdade, um resultado geral, chamado Teorema de Rice, enuncia que testar *qualquer propriedade* das linguagens reconhecidas por máquinas de Turing é indecidível. Damos o Teorema de Rice no Problema 5.22.

Até agora, nossa estratégia para provar que linguagens são indecidíveis envolve uma redução a partir de A_{MT} . Às vezes reduzir a partir de alguma outra linguagem indecidível, tal como V_{MT} , é mais conveniente quando estamos mostrando que certas linguagens são indecidíveis. O teorema a seguir mostra que testar a equivalência de duas máquinas de Turing é um problema indecidível. Poderíamos prová-lo por uma redução a partir de A_{MT} , mas usamos essa oportunidade para dar um exemplo de uma prova de indecidibilidade por redução a partir de V_{MT} . Seja

$$EQ_{MT} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ e } M_2 \text{ são MT's e } L(M_1) = L(M_2)\}.$$

Teorema 5.4

EQ_{MT} é indecidível.

.....

Idéia da prova. Mostre que, se EQ_{MT} fosse decidível, V_{MT} também seria decidível, dando uma redução de V_{MT} para EQ_{MT} . A idéia é simples. V_{MT} é o problema de se testar se a linguagem de uma MT é vazia. EQ_{MT} é o problema de se testar se as linguagens de duas MT's são a mesma. Se uma dessas linguagens acontece de ser \emptyset , acabamos com o problema de se testar se a linguagem da outra máquina é vazia, ou seja, o problema V_{MT} . Portanto, em um certo sentido, o problema V_{MT} é um caso especial do problema EQ_{MT} no qual uma das máquinas está fixada para reconhecer a linguagem vazia. Essa idéia faz com que dar a redução fique fácil.

Prova. Supomos que a MT R decide EQ_{MT} e construimos a MT S para decidir V_{MT} da seguinte maneira.

S = “Sobre a entrada $\langle M \rangle$, onde M é uma MT:

1. Roder R sobre a entrada $\langle M, M_1 \rangle$, onde M_1 é uma MT que rejeita todas as entradas.
2. Se R aceita, *aceite*; se R rejeita, *rejeite*.

Se R decide EQ_{MT} , S decide V_{MT} . Mas V_{MT} é indecidível pelo Teorema 5.2, portanto EQ_{MT} também tem que ser indecidível.

Reduções via histórias de computação

O método da história de computação é uma técnica importante para provar que A_{MT} é redutível a certas linguagens. Esse método é frequentemente útil quando o problema a ser mostrado indecidível envolve testar pela existência de algo. Por exemplo, esse método é usado para mostrar a indecidibilidade do décimo problema de Hilbert, testar pela existência de raízes inteiras em um polinômio.

A história de computação para uma máquina de Turing sobre uma entrada é simplesmente a seqüência de configurações pelas quais que a máquina passa quando ela processa a entrada. É um registro completo da computação dessa máquina.

Definição 5.5

Seja M uma máquina de Turing e w uma cadeia de entrada. Uma **história de computação de aceitação** para M sobre w é uma seqüência de configurações C_1, C_2, \dots, C_l , onde C_1 é a configuração inicial de M sobre w , C_l é uma configuração de aceitação de M , e cada C_i legitimamente segue de C_{i-1} conforme as regras de M . Uma **história de computação de rejeição** para M sobre w é definida similarmente, exceto que C_l é uma configuração de rejeição.

Histórias de computação são seqüências finitas. Se M não pára sobre w , nenhuma história de computação de aceitação ou de rejeição existe para M sobre w . Máquinas determinísticas têm no máximo uma história de computação sobre qualquer entrada. Máquinas não-determinísticas podem ter muitas histórias de computação sobre uma única entrada, correspondendo aos vários ramos da computação. Por enquanto, continuamos a nos concentrar em máquinas determinísticas. Nossa primeira prova de indecidibilidade usando o método da história de computação concerne um tipo de máquina chamada autômato linearmente limitado.

Definição 5.6

Um **autômato linearmente limitado** é um tipo restrito de máquina de Turing no qual a cabeça da fita não pode mover para fora da parte da fita contendo a entrada. Se a máquina tentar mover sua cabeça para fora de qualquer uma das extremidades da entrada, a cabeça permanece onde está, da mesma maneira que a cabeça não moverá para fora da extremidade esquerda da fita de uma máquina de Turing comum.

Um autômato linearmente limitado é uma máquina de Turing com uma quantidade limitada de memória, como mostrado esquematicamente na Figura 5.1. Ele pode resolver somente problemas que requerem memória que pode caber dentro da fita usada para a entrada. Usando um alfabeto de fita maior que o alfabeto de entrada permite a memória disponível a ser aumentada de até um fator constante. Daí dizemos que para uma entrada de comprimento n , a quantidade de memória é linear em n —daí o nome desse modelo.

Figura 5.1: Esquema de um autômato linearmente limitado

A despeito de sua restrição de memória, autômatos linearmente limitados são bastante poderosos. Por exemplo, os decisores para A_{AFD} , A_{GLC} , V_{AFD} , e V_{GLC} todos são ALL's. Toda LLC pode ser decidida por um ALL. Na verdade, chegar a uma linguagem decidível que não pode ser decidida por um ALL dá algum trabalho. Desenvolvemos as técnicas para fazer isso no Capítulo 9.

Aqui, A_{ALL} é o problema de se testar se um ALL aceita sua entrada. Muito embora A_{ALL} seja o mesmo que o problema indecidível A_{MT} onde a máquina de Turing é restrita a ser um ALLm podemos mostrar que A_{ALL} é decidível. Seja

$$A_{\text{ALL}} = \{\langle M, w \rangle \mid M \text{ é um ALL que aceita a cadeia } w\}.$$

Antes de provar a decidibilidade de A_{ALL} , achamos que o lema a seguir é útil. Ele diz que um ALL pode ter somente um número limitado de configurações quando uma cadeia de comprimento n é a entrada.

Lema 5.7

Seja M um ALL com q estados e g símbolos no alfabeto de símbolo. Existem exatamente qng^n configurações distintas de M para uma fita de comprimento n .

Prova. Lembremo-nos de que uma configuração de M é como um retrato instantâneo no meio de sua computação. Uma configuração consiste do estado do controle, posição da cabeça, e conteúdo da fita. Aqui, M tem q estados. O comprimento de sua fita é n , portanto a cabeça pode estar em uma das n posições, e g^n possíveis cadeias de símbolos de fita aparecem na fita. O produto dessas três quantidades é o número total de diferentes configurações de M com uma fita de comprimento n .

Teorema 5.8

A_{ALL} é decidível.

Idéia da prova. De modo a decidir se o ALL M aceita a entrada w , simulamos M sobre w . No decorrer da simulação, se M pára e aceita ou rejeita, aceitamos ou rejeitamos conformemente. A dificuldade ocorre se M entra em loop sobre w . Precisamos de ser capazes de detectar a entrada em loop de modo que possamos parar e rejeitar.

A idéia para detectar quando M está em loop é que, à medida M que computa sobre w , ela continua a repetir essa configuração muitas e muitas vezes e por conseguinte está em loop. Devido ao fato de M é um ALL, a quantidade de fita que lhe é disponível é limitada. Pelo Lema 5.7, M pode estar em somente um número limitado de configurações sobre essa quantidade de fita. Portanto somente uma quantidade limitada de tempo está disponível a M antes que ela entre em alguma configuração em que ela tenha entrado anteriormente. Detectar que M está em loop é possível simulando M para o número de passos dado pelo Lema 5.7. se M não parou até aí, ela tem que estar em loop.

Prova. O algoritmo que decide A_{ALL} é o seguinte.

$L =$ “Sobre a entrada $\langle M, w \rangle$, onde M é um ALL e w é uma cadeia:

1. Simule M sobre w por qng^n passos ou até que ela páre.
2. Se M parou, *aceite* se ela aceitou e *rejeite* se ela rejeitou. Se ela não parou, *rejeite*.”

Se M sobre w não parou dentro de qng^n passos, ela tem que estar repetindo uma configuração de acordo com o Lema 5.7 e portanto em loop. É por isso que nosso algoritmo rejeita nessa instância.

O Teorema 5.8 mostra que ALL's e MT's diferem de uma maneira essencial: Para ALL's o problema da aceitação é decidível, mas para MT's não o é. Entretanto, alguns outros problemas envolvendo ALL's permanecem indecidíveis. Uma é o problema da vacuidade $V_{\text{ALL}} = \{\langle M \rangle \mid M \text{ é um ALL onde } L(M) = \emptyset\}$. Para provar que V_{ALL} é indecidível, damos uma redução que usa o método da história de computação.

Teorema 5.9

V_{ALL} é indecidível.

Idéia da prova. Esta prova é por redução a partir de A_{MT} . Mostramos que, se V_{ALL} fosse decidível, A_{MT} também seria. Suponha que V_{ALL} seja decidível. Como podemos usar essa suposição para decidir A_{MT} ?

Para uma MT M e uma entrada w podemos determinar se M aceita w construindo um certo ALL B e aí testar se $L(B)$ é vazia. A linguagem que B reconhece compreende todas as histórias de computação para M sobre w . Se M aceita w , essa linguagem contém uma cadeia e portanto é não-vazia. Se M não aceita w , essa linguagem é vazia. Se podemos determinar se a linguagem de B é vazia, claramente podemos determinar se M aceita w .

Agora descrevemos como construir B partindo de M e w . Note que precisamos mostrar mais que a mera existência de B . Temos que mostrar como uma máquina de Turing pode obter uma descrição de B dadas as descrições de M e w .

Construimos B para aceitar sua entrada x se x é uma história de computação de aceitação para M sobre w . Relembremo-nos que uma história de computação de aceitação é a seqüência de configurações, C_1, C_2, \dots, C_l pela qual M passa quando ela aceita alguma cadeia w . Para os propósitos desta prova assumimos que a história de computação de aceitação é apresentada como uma única cadeia, com as configurações separadas uma da outra pelo símbolo $\#$, como mostrado na Figura 5.2.

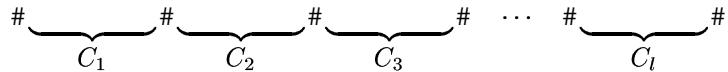


Figura 5.2: Uma possível entrada para B

O ALL B funciona da seguinte maneira. Quando ele recebe uma entrada x , B supostamente aceita x se x é uma computação de aceitação para M sobre w . Primeiro, B quebra x conforme os delimitadores em cadeias C_1, C_2, \dots, C_l . Então B verifica se a C_i satisfaz as três condições de uma história de computação:

1. C_1 é uma configuração inicial para M sobre w .
2. Cada C_{i+1} legítimamente segue de C_i .
3. C_l é uma configuração de aceitação para M .

A configuração inicial C_1 para M sobre w é a cadeia $q_0w_1w_2 \dots w_n$, onde q_0 é o estado inicial para M sobre w . Aqui, B tem essa cadeia diretamente embutida, de modo que ela seja capaz de verificar a primeira condição. Uma configuração de aceitação é uma que contém o estado q_{aceita} , portanto B pode verificar a terceira condição fazendo uma varredura em C_l por q_{aceita} . A segunda condição é a mais difícil de verificar. Para cada par de configurações adjacentes, B verifica se C_{i+1} legítimamente segue de C_i . Esse passo envolve verificar que C_i e C_{i+1} são idênticas exceto pelas posições sob e adjacente à cabeça em C_i . Essas posições têm que ser atualizadas conforme a função

de transição de M . Então, B verifica que a atualização foi feita corretamente fazendo um zigue-zague entre posições correspondentes de C_i e C_{i+1} . Para manter o registro das posições correntes enquanto faz o zigue-zague, B marca a posição corrente com pontos sobre a fita. Finalmente, se as condições 1, 2, e 3 são satisfeitas, B aceita sua entrada.

Note que o ALL B não é construído para os propósitos de verdadeiramente rodá-lo sobre alguma entrada—uma confusão comum. Construimos B somente para o propósito de alimentar uma descrição de B num decisor para V_{ALL} que assumimos existir. Uma vez que esse decisor retorna sua resposta podemos invertê-la para obter a resposta se M aceita w . Por conseguinte podemos decidir A_{MT} , uma contradição.

Prova. Agora estamos prontos para enunciar a redução de A_{MT} para V_{ALL} . Suponha que a MT R decida V_{ALL} . Construa a MT S que decide A_{MT} da seguinte forma.

S = “Sobre a entrada $\langle M, w \rangle$, onde M é uma MT e w é uma cadeia:

1. Construa o ALL B a partir de M e w conforme descrito na idéia da prova.
2. Rode R sobre a entrada $\langle B \rangle$.
3. Se R rejeita, *aceite*; se R aceita, *rejeite*.”

Figura 5.3: O ALL B verificando uma história de computação de uma MT

Se R aceita $\langle B \rangle$, então $L(B) = \emptyset$. Por conseguinte M não tem história de computação de aceitação sobre w e M não aceita w . Consequentemente S rejeita $\langle M, w \rangle$. Similarmente se R rejeita $\langle B \rangle$, a linguagem de B é não-vazia. A única cadeia que B pode aceitar é uma história de computação de aceitação para M sobre w . Portanto M tem que aceitar w . Consequentemente, S aceita $\langle M, w \rangle$. A Figura 5.3 mostra tal verificação de uma história de computação de uma MT.

Podemos também usar a técnica de redução via histórias de computação para estabelecer a indecidibilidade de certos problemas relacionados a gramáticas livres-do-contexo e autômatos a pilha. Relembremo-nos de que no Teorema 4.7 apresentamos um algoritmo para decidir se uma gramática livre-do-contexto gera alguma cadeia, ou seja, se $L(G) \neq \emptyset$. Agora mostramos que um problema relacionado é indecidível. É o problema de se testar se uma gramática livre-do-contexto gera todas as cadeias possíveis. Provar que esse problema é indecidível é o passo principal na demonstração de que o problema da equivalência para gramáticas livres-do-contexto é indecidível. Seja

$$TODAS_{\text{GLC}} = \{\langle G \rangle \mid G \text{ é uma GLC e } L(G) = \Sigma^*\}.$$

Teorema 5.10

$TODAS_{\text{GLC}}$ é indecidível.

Prova. Esta prova é por contradição. Para obter a contradição assumimos que $TODAS_{GLC}$ é decidível e usamos essa suposição para mostrar que A_{MT} é decidível. Esta prova é semelhante àquela do Teorema 5.9 mas com um pequeno truque extra: Ela é uma redução de A_{MT} via histórias de computação, mas temos que modificar a representação das histórias de computação levemente por uma razão técnica que explicamos mais adiante.

Agora descrevemos como usar um procedimento de decisão para $TODAS_{GLC}$ para decidir A_{MT} . Para uma MT M e uma entrada w construímos uma GLC G que gera todas as cadeias se e somente se M não aceita w . Portanto, se M não aceita w , G não gera alguma cadeia específica. Essa cadeia é—adivinhe—a história de computação de aceitação para M sobre w . Ou seja, G é projetada para gerar todas as cadeias que não são histórias de computação de aceitação para M sobre w .

Para fazer a GLC G gerar todas as cadeias que falhem em ser uma história de computação de aceitação para M sobre w , utilizamos a seguinte estratégia. Uma cadeia pode falhar em ser uma história de computação de aceitação por várias razões. Uma história de computação de aceitação para M sobre w aparece como $\#C_1\#C_2\#\dots\#C_l\#$, onde C_i é a história de computação para M no i -ésimo passo da computação sobre w . Então G gera todas as cadeias que

1. não começam com C_1 ,
2. não terminam com uma configuração de aceitação, ou
3. onde algum C_i não leva corretamente a C_{i+1} sob as regras de M .

Se M não aceita w , nenhuma história de computação de aceitação existe, portanto *todas* as cadeias falham de uma maneira ou de outra. Por conseguinte G geraria todas as cadeias, como desejado.

Agora vamos à verdadeira construção de G . Ao invés de construir G , construímos um AP D . Sabemos que podemos usar a construção dada no Teorema 2.12 na página 106 para converter D para uma GLC. Fazemos isso porque, para nossos propósitos, projetar um AP é mais fácil que projetar uma GLC. Nesta instância, D começará ramicando não-deterministicamente para adivinhar qual das três condições anteriores verificar. Um ramo verifica se o começo da cadeia de entrada é C_1 e aceita se não o é. Um outro ramo verifica se a cadeia de entrada termina com uma configuração contendo o estado de aceitação, q_{aceita} , aceita se não o é.

O terceiro ramo deverá aceitar se alguma C_i não leva corretamente a C_{i+1} . Ele funciona fazendo uma varredura na entrada até que ele não-deterministicamente decide que ela veio para C_i . A seguir, ele empilha C_i até que ele chega no final marcado pelo símbolo $\#$. Então, D desempilha para comparar com C_{i+1} . Elas deveriam casar exceto em torno da posição da cabeça onde a diferença é ditada pela função de transição de M . Finalmente, D aceita se é descasamento ou uma atualização incorreta.

O problema com essa idéia é que, quando D desempilha C_i , ela está na ordem reversa e não está adequada para comparação com C_{i+1} . Nesse ponto o truque na prova aparece: Escrevemos uma história de computação de aceitação diferentemente. Alternadamente, uma em cada configuração aparece na ordem reversa. As posições ímpares permanecem escritas na ordem de-trás-para-a-frente, mas as posições pares são escritas de-frente-para-trás. Por conseguinte uma história de computação de aceitação apareceria como mostrado na Figura 5.4.

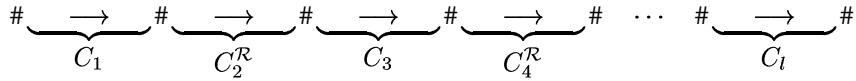


Figura 5.4: Alternadamente, uma em cada duas configurações escritas em ordem reversa

Nessa forma modificada o AP é capaz de empilhar uma configuração de modo que quando ela é desempilhada a ordem está adequada para comparação com a próxima. Projetamos D para aceitar qualquer cadeia que não é uma história de computação de aceitação na forma modificada.

5.2 Um problema indecidível simples.....

Nesta seção mostramos que o fenômeno da indecidibilidade não está confinado a problemas concernentes a autômatos. Damos um exemplo de um problema indecidível concernente a simples manipulações de cadeias. É chamado o *problema da correspondência de Post*, ou *PCP*.

Podemos descrever esse problema facilmente como um tipo de charada. Começamos com uma coleção de dominós, cada um combinando duas cadeias, uma em cada lado. Um dominó individual tem a seguinte aparência

$$\left[\begin{array}{c} a \\ ab \end{array} \right]$$

e uma coleção de dominós é da forma

$$\left\{ \left[\begin{array}{c} b \\ ca \end{array} \right], \left[\begin{array}{c} a \\ ab \end{array} \right], \left[\begin{array}{c} ca \\ a \end{array} \right], \left[\begin{array}{c} abc \\ c \end{array} \right] \right\}$$

O trabalho é fazer uma lista desses dominós (repetições permitidas) de modo que a cadeia que obtemos lendo os símbolos em cima é a mesma que a cadeia de símbolos embaixo. Essa lista é chamada um *emparelhamento*. Por exemplo, a seguinte lista é um emparelhamento para essa charada.

$$\left[\begin{array}{c} a \\ ab \end{array} \right] \left[\begin{array}{c} b \\ ca \end{array} \right] \left[\begin{array}{c} ca \\ a \end{array} \right] \left[\begin{array}{c} a \\ ab \end{array} \right] \left[\begin{array}{c} abc \\ c \end{array} \right].$$

Lendo a cadeia de cima obtemos $abcaaabc$, que é a mesma ao ler a de baixo. Podemos ilustrar esse emparelhamento deformando os dominós de modo que os símbolos correspondentes de cima e de baixo se alinhem.

Para algumas coleções de dominós encontrar um emparelhamento pode não ser possível. Por exemplo, a coleção

$$\left\{ \left[\begin{array}{c} abc \\ ab \end{array} \right], \left[\begin{array}{c} ca \\ a \end{array} \right], \left[\begin{array}{c} acc \\ ba \end{array} \right] \right\}$$

não pode conter um emparelhamento porque toda cadeia de cima é maior que a cadeia de baixo correspondente.

O problema da correspondência de Post é determinar se uma coleção de dominós tem um emparelhamento. Esse problema é insolúvel por algoritmos.

Antes de chegar ao enunciado formal desse teorema e sua prova, vamos enunciar o problema precisamente e então expressá-lo como uma linguagem. Uma instância do PCP é uma coleção P de dominós

$$P = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\},$$

e um emparelhamento é uma seqüência i_1, i_2, \dots, i_l , onde $t_{i_1} t_{i_2} \dots t_{i_l} = b_{i_1} b_{i_2} \dots b_{i_l}$. O problema é determinar se P tem um emparelhamento. Seja

$$\text{PCP} = \{ \langle P \rangle \mid P \text{ é uma instância do problema da correspondência de Post com um emparelhamento} \}.$$

Teorema 5.11
 PCP é indecidível.

Idéia da prova. Conceitualmente esta prova é simples, embora tenha muitos detalhes técnicos. A técnica principal é redução a partir de A_{MT} via histórias de computação de aceitação. Mostramos que de qualquer MT M e entrada w podemos construir uma instância P onde um emparelhamento é uma história de computação de aceitação para M sobre w . Se pudéssemos determinar se a instância tem um emparelhamento, seríamos capazes de determinar se M aceita w .

Como podemos construir P de modo que o emparelhamento seja uma história de computação de aceitação para M sobre w ? Escolhemos os dominós em P de modo que fazer um emparelhamento force que uma simulação de M ocorra. No emparelhamento, cada dominó liga uma posição ou posições em uma configuração com a(s) correspondente(s) na próxima configuração.

Antes de chegar à construção tratamos dois pequenos detalhes técnicos. (Não se preocupe demais com eles na sua primeira leitura dessa construção.) Primeiro, por conveniência em construir P , assumimos que M sobre w nunca tenta mover sua cabeça para fora da extremidade esquerda da fita. Isso requer primeiro alterar M para evitar esse comportamento. Segundo, modificamos o PCP para requerer que um emparelhamento comece com o primeiro dominó

$$\left[\frac{t_1}{b_1} \right].$$

Mais adiante mostramos como eliminar esse requisito. Chamamos esse problema o problema da correspondência de Post modificado, PCPM. Seja

$$\text{PCPM} = \{ \langle P \rangle \mid P \text{ é uma instância do problema da correspondência de Post com um emparelhamento que começa com o primeiro dominó} \}.$$

Agora vamos aos detalhes da prova e projetar P para simular M sobre w .

Prova. Supomos que a MT R decide o PCP e construímos S que decide A_{MT} . Seja

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{aceita}}, q_{\text{rejeita}}),$$

onde Q , Σ , Γ , e δ , são o conjunto de estados, o alfabeto de entrada, o alfabeto de fita, e a função de transição de M , respectivamente.

Nesse caso S constrói uma instância do PCP P que tem um emparelhamento se e somente se M aceita w . Para fazer isso S primeiro constrói uma instância P' do PCPM. Descrevemos a construção em sete partes, cada uma das quais realiza um aspecto específico de se simular M sobre w . Para explicar o que estamos fazendo intercalamos a construção com um exemplo da construção em ação.

Parte 1. A construção começa da seguinte maneira

$$\text{Ponha } \left[\frac{\#}{\#q_0w_1w_2 \cdots w_n\#} \right] \text{ em } P' \text{ como o primeiro dominó } \left[\frac{t_1}{b_1} \right].$$

Devido ao fato de que P' é uma instância do PCPM o emparelhamento tem que começar com esse dominó. Por conseguinte a cadeia de baixo começa corretamente com $C_1 = q_0w_1w_2 \cdots w_n$, a primeira configuração na história de computação de aceitação para M sobre w , como mostrado na Figura 5.5.

Figura 5.5: Início do emparelhamento do PCPM

Nessa ilustração do emparelhamento parcial obtido até então, a cadeia de baixo consiste de $\#q_0w_1w_2 \cdots w_n\#$ e a cadeia de cima consiste apenas de $\#$. Para obter um emparelhamento precisamos estender a cadeia de cima para emparelhar com a cadeia de baixo. Fornecemos dominós adicionais para permitir essa extensão. Os dominós adicionais fazem com que a próxima configuração de M apareça na extensão da cadeia de baixo forçando uma simulação de um único-passo de M .

Nas partes 2, 3, e 4, adicionamos a P' dominós que realizam a parte principal da simulação. A parte 2 lida com os movimentos da cabeça para a direita, a parte 3 lida com os movimentos da cabeça para a esquerda, e a parte 4 lida com as células da fita não adjacentes à cabeça.

Parte 2. Para todos $a, b \in \Gamma$ e todos $q, r \in Q$ onde $q \neq q_{\text{rejeita}}$,

$$\text{se } \delta(q, a) = (r, b, D), \text{ ponha } \left[\frac{qa}{br} \right] \text{ em } P'.$$

Parte 3. Para todos $a, b, c \in \Gamma$ e todos $q, r \in Q$ onde $q \neq q_{\text{rejeita}}$,

$$\text{se } \delta(q, a) = (r, b, E), \text{ ponha } \left[\frac{cqa}{rcb} \right] \text{ em } P'.$$

Parte 4. Para todo $a \in \Gamma$,

$$\text{ponha } \left[\frac{a}{a} \right] \text{ em } P'.$$

Agora criamos um exemplo hipotético para ilustrar o que construímos até agora. Seja $\Gamma = \{0, 1, 2, \sqcup\}$. Digamos que w é a cadeia 0100 e que o estado inicial de M é q_0 . No estado q_0 , ao ler um 0, digamos que a função de transição ordena que M

entre no estado q_7 , escreva um 2 na fita, e move sua cabeça para a direita. Em outras palavras, $\delta(q_0, 0) = (q_7, 2, D)$.

A parte 1 coloca o dominó

$$\left[\frac{\#}{\#q_0 0100} \right] = \left[\frac{t_1}{b_1} \right]$$

em P' , e o emparelhamento começa:

(figura 1 da pagina 186)

Adicionalmente, a parte 2 coloca o dominó

$$\left[\frac{q_0 0}{2q_7 r} \right]$$

pois $\delta(q_0, 0) = (q_7, 2, D)$ e a parte 4 coloca o dominó

$$\left[\frac{0}{0} \right], \left[\frac{1}{1} \right], \left[\frac{2}{2} \right], \text{ e } \left[\frac{\sqcup}{\sqcup} \right]$$

em P' , pois 0, 1, 2, e \sqcup são os membros de Γ . Isso, juntamente com a parte 5, nos permite estender o emparelhamento da seguinte forma:

(figura 2 da pagina 186)

Por conseguinte os dominós das partes 2,3, e 4 nos deixam estender o emparelhamento adicionando a segunda configuração após a primeira. Queremos que esse processo continue, adicionando a terceira configuração, e aí a quarta, e assim por diante. Para isso acontecer precisamos adicionar um dominó a mais para copiar o símbolo #.

Parte 5.

Ponha $\left[\frac{\#}{\#} \right]$ e $\left[\frac{\#}{\sqcup \#} \right]$ em P' .

O primeiro desses dominós nos permite copiar o símbolo # que marca a separação das configurações. Além disso, o segundo dominó nos permite adicionar um símbolo branco \sqcup no final da configuração para simular a quantidade infinita de brancos à direita que são suprimidos quando escrevemos a configuração.

Continuando com o exemplo, vamos dizer que no estado q_7 , ao ler um 1 M vai para o estado q_5 , escreve um 0, e move a cabeça para a direita. Ou seja, $\delta(q_7, 1) = (q_5, 0, D)$. Então temos o dominó

$$\left[\frac{q_7 1}{0 q_5} \right] \text{ em } P'.$$

Portanto o último emparelhamento parcial se estende para

(figura 1 da pagina 187)

Então, suponha que no estado q_5 , ao ler um 0, M vai para o estado q_9 , escreve um 2, e move sua cabeça para a esquerda. Portanto $\delta(q_5, 0) = (q_9, 2, E)$. Então temos os dominós

$$\left[\frac{0 q_5 0}{q_9 0 2} \right], \left[\frac{1 q_5 0}{q_9 1 2} \right], \left[\frac{2 q_5 0}{q_9 2 2} \right], \text{ e } \left[\frac{\sqcup q_5 0}{q_9 \sqcup 2} \right].$$

O primeiro é relevante porque o símbolo à esquerda da cabeça é um 0. O emparelhamento parcial precedente se estende para

(figura 2 da pagina 187)

Note que, à medida que construímos um emparelhamento, somos forçados a simular M sobre a entrada w . Esse processo continua até que M atinja um estado de parada. Se um estado de aceitação ocorre, queremos fazer com que a parte de cima do emparelhamento parcial “acompanhe” a parte de baixo de modo que o emparelhamento seja completo. Podemos arranjar para que isso aconteça adicionando mais dominós.

Parte 6. Para todo $a \in \Gamma$,

$$\text{ponha } \left[\frac{aq_{\text{aceita}}}{q_{\text{aceita}}} \right] \text{ e } \left[\frac{q_{\text{aceita}}a}{q_{\text{aceita}}} \right] \text{ em } P'.$$

Esse passo tem o efeito de adicionar “pseudo-passos” da máquina de Turing depois que ela tenha parado, onde a cabeça “come” símbolos adjacentes até que nenhum mais permaneça. Continuando com o exemplo anterior, se o emparelhamento parcial até o ponto quando a máquina pára em um estado de aceitação é

(figura 1 da pagina 188)

Os dominós que acabamos de acrescentar permitem que o emparelhamento continue:

(figura 2 da pagina 188)

Parte 7. Finalmente adicionamos o dominó

$$\left[\frac{q_{\text{aceita}}\#\#}{\#} \right]$$

e completamos o emparelhamento:

(figura 3 da pagina 188)

Isso conclui a construção de P' . Lembre-se que P' é uma instância do PCPM na qual o emparelhamento simula a computação de M sobre w . Para terminar a prova, lembremo-nos que o PCPM difere do PCP no sentido de que o emparelhamento tem que começar com o primeiro dominó na lista. Se olharmos para P' como uma instância do PCP ao invés do PCPM, ele obviamente contém um emparelhamento, independentemente de se M pára sobre w . Você pode vê-lo? (Dica: É muito curto.)

Agora mostramos como converter P' em P , uma instância do PCP que ainda simula M sobre w . Fazemos isso com um truque um tanto técnico. A idéia é construir o requisito de começar com o primeiro dominó direto no problema do modo que enunciar o requisito fica desnecessário. Precisamos introduzir um pouco de notação para esse propósito.

Seja $u = u_1 u_2 \cdots u_n$ uma cadeia qualquer de comprimento n . Defina $\star u$, $u\star$, e $\star u\star$ como sendo as cadeias

$$\begin{aligned} \star u &= *u_1 * u_2 * u_3 * \cdots * u_n \\ u\star &= u_1 * u_2 * u_3 * \cdots * u_n * \\ \star u\star &= *u_1 * u_2 * u_3 * \cdots * u_n *. \end{aligned}$$

Aqui, $\star u$ adiciona o símbolo $*$ antes de cada caracter em u , $u\star$ adiciona um após cada caracter em u , e $\star u\star$ adiciona um tanto quanto depois de cada caracter em u .

Para converter P' em P , uma instância do PCP, fazemos o seguinte. Se P' fosse a coleção

$$\left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \left[\frac{t_3}{b_3} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\},$$

fazemos P ser a coleção

$$\left\{ \left[\frac{\star t_1}{\star b_1 \star} \right], \left[\frac{\star t_1}{b_1 \star} \right], \left[\frac{\star t_2}{b_2 \star} \right], \left[\frac{\star t_3}{b_3 \star} \right], \dots, \left[\frac{\star t_k}{b_k \star} \right], \left[\frac{\star \Diamond}{\Diamond} \right] \right\}.$$

porque ela é a única na qual ambas as partes de cima e de baixo começam com o mesmo símbolo, a saber \star . Além de forçar que o emparelhamento comece com o primeiro dominó, a presença dos \star 's não afeta possíveis emparelhamentos porque eles simplesmente se intercalam com os símbolos originais. Os símbolos originais agora ocorrem nas posições pares do emparelhamento. O dominó

$$\left[\frac{\star \Diamond}{\Diamond} \right]$$

está aí para permitir que a parte de cima acrescente o \star a mais no final do emparelhamento.

.....

5.3 Redutibilidade por mapeamento

Mostramos como usar a técnica da redutibilidade para provar que vários problemas são indecidíveis. Nesta seção formalizamos a noção de redutibilidade. Fazer isso nos permite usar redutibilidade de maneiras mais refinadas, tais como provar que certas linguagens não são Turing-reconhecíveis e para aplicações em teoria da complexidade.

A noção de reduzir um problema a outro pode ser definida formalmente de uma dentre diversas maneiras. A escolha de qual delas usar depende da aplicação. Nossa escolha é um tipo simples de redutibilidade chamado **redutibilidade por mapeamento**.²

A grosso modo, ser capaz de reduzir o problema A ao problema B usando uma redutibilidade por mapeamento significa que uma função computável existe que converte instâncias do problema A para instâncias do problema B . Se tivermos tal função de conversão, chamado de uma *redução*, podemos resolver A com um resolvedor para B . A razão é que qualquer instância de A pode ser resolvida primeiro usando uma redução para convertê-la numa instância de B e aí aplicar o resolvedor para B . Uma definição precisa de redutibilidade por mapeamento segue em breve.

Funções computáveis

Uma máquina de Turing computa uma função iniciando com a entrada para a função sobre a fita e parando com a saída da função na fita.

Definição 5.12

Uma função $f : \Sigma^* \rightarrow \Sigma^*$ é uma **função computável** se alguma máquina de Turing M , sobre toda entrada w , pára com exatamente $f(w)$ na sua fita.

²Ele é chamado **redutibilidade muitos-para-um** em alguns outros livros-texto.

Exemplo 5.13

Todas as operações aritméticas usuais sobre inteiros são funções computáveis. Por exemplo, podemos fazer uma máquina que toma como entrada $\langle m, n \rangle$ e retorna $m + n$, a soma de m e n . Não damos quaisquer detalhes aqui, deixando-os como exercícios.

Exemplo 5.14

Funções computáveis podem ser transformações de descrições de máquinas. Por exemplo, uma função computável f toma como entrada w e retorna a descrição de uma máquina de Turing $\langle M' \rangle$ se $w = \langle M \rangle$ é uma codificação de uma máquina de Turing M . A máquina M' é uma máquina que reconhece a mesma linguagem que M , mas nunca tenta mover sua cabeça para fora da extremidade esquerda de sua fita. A função f realiza essa tarefa adicionando vários estados à descrição de M . A função retorna ε se w não é uma codificação legítima de uma máquina de Turing.

Definição formal de redutibilidade por mapeamento

Agora definimos redutibilidade por mapeamento. Como de costume representamos problemas computacionais por linguagens.

Definição 5.15

A linguagem A é **redutível por mapeamento** à linguagem B , escrito $A \leq_m B$, se existe uma função computável $f : \Sigma^* \rightarrow \Sigma^*$, onde para toda w ,

$$w \in A \iff Bf(w) \in B.$$

A função f é chamada a **redução** de A para B .

A Figura 5.6 ilustra a redutibilidade por mapeamento.

Figura 5.6: A função f reduzindo A para B

Uma redução por mapeamento de A para B provê uma maneira de converter questões sobre teste de pertinência em A para teste de pertinência em B . Para testar se $w \in A$, usamos a redução f para mapear w para $f(w)$ e testar se $f(w) \in B$. O termo *redução por mapeamento* vem da função ou mapeamento que provê os meios de se fazer a redução.

Se um problema é redutível por mapeamento a um segundo, que já tenha sido previamente resolvido, podemos portanto obter uma solução para o problema original. Capturamos essa idéia no teorema seguinte.

Teorema 5.16

Se $A \leq_m B$ e B é decidível, então A é decidível.

Prova. Fazemos M ser o decisor para B e f a redução de A para B . Descrevemos um decisor N para A da seguinte forma.

N = “Sobre a entrada w :

1. Compute $f(w)$.
2. Rode M sobre a entrada $f(w)$ e dê como saída o que quer que M dê como saída.”

Claramente, se $w \in A$, então $f(w) \in B$ porque f é uma redução de A para B . Por conseguinte M aceita $f(w)$ sempre que $w \in A$. Daí N funciona como desejado.

O corolário seguinte do Teorema 5.16 tem sido nossa principal ferramenta para provar indecidibilidade.

Corolário 5.17

Se $A \leq_m B$ e A é indecidível, então B é indecidível.

Agora revisitamos algumas de nossas provas anteriores que usaram o método da redutibilidade para obter exemplos de redutibilidades por mapeamento.

Exemplo 5.18

No Teorema 5.1 usamos uma redução de A_{MT} para provar que PARA_{MT} é indecidível. Essa redução mostrou como um decisor para PARA_{MT} poderia ser usado para dar um decisor para A_{MT} . Podemos demonstrar uma redutibilidade por mapeamento de A_{MT} para PARA_{MT} como segue. Para fazer isso temos que apresentar uma função f que toma uma entrada da forma $\langle M, w \rangle$ e retorna uma saída da forma $\langle M', w' \rangle$, onde

$$\langle M, w \rangle \in A_{\text{MT}} \text{ se e somente se } \langle M', w' \rangle \in \text{PARA}_{\text{MT}}.$$

A seguinte máquina F computa uma redução f .

F = “Sobre a entrada $\langle M, w \rangle$:

1. Construa a seguinte máquina M' .

M' = “Sobre a entrada x :

1. Rode M sobre x .
2. Se M aceita, *aceite*.
3. Se M rejeita, entre em loop.”

2. Dê como saída $\langle M', w' \rangle$.”

Exemplo 5.19

A prova da indecidibilidade do problema da correspondência de Post no Teorema 5.11 contém duas reduções por mapeamento. Primeiro, ela mostra que $A_{\text{MT}} \leq_m \text{PCPM}$ e então ela mostra que $\text{PCPM} \leq_m \text{PCP}$. Em ambos os casos podemos facilmente obter a verdadeira função de redução e mostrar que ela é uma redução por mapeamento. Como o Exercício 5.6 mostra, redutibilidade por mapeamento é transitiva, de modo que essas duas reduções juntas implicam que $A_{\text{MT}} \leq_m \text{PCP}$.

Exemplo 5.20

Uma redução por mapeamento de V_{MT} para EQ_{MT} está por trás da prova do Teorema 5.4. Nesse caso a redução f mapeia a entrada $\langle M \rangle$ para a saída $\langle M, M_1 \rangle$, onde M_1 é a máquina que rejeita todas as entradas.

Exemplo 5.21 dotfill

A prova do Teorema 5.2 mostrando que V_{MT} é indecidível ilustra a diferença entre a noção formal de redutibilidade por mapeamento que definimos nesta seção e a noção informal de redutibilidade que usamos anteriormente neste capítulo. A prova mostra que V_{MT} é indecidível reduzindo A_{MT} para ele. Vamos ver se podemos converter essa redução em uma redutibilidade por mapeamento.

Da redução original podemos facilmente construir uma função f que toma como entrada $\langle M, w \rangle$ e produz a saída $\langle M_1 \rangle$, onde M_1 é a máquina de Turing descrita naquela prova. Mas M aceita w se e somente se $L(M_1)$ não é vazia portanto f é uma redução por mapeamento de A_{MT} para $\overline{V_{\text{MT}}}$. Ela ainda mostra que V_{MT} é indecidível porque decidibilidade não é afetada por complementação, mas ela não dá uma redução por mapeamento de A_{MT} para V_{MT} . Na verdade, nenhuma tal redução existe, como lhe pede para mostrar o Exercício 5.5.

A sensitividade da redutibilidade por mapeamento à complementação é importante no uso da redutibilidade para provar não-reconhecibilidade de certas linguagens. Podemos também usar redutibilidade por mapeamento para mostrar que problemas não são Turing-reconhecíveis. O teorema seguinte é análogo ao Teorema 5.16.

Teorema 5.22

Se $A \leq_m B$ e B é Turing-reconhecível, então A é Turing-reconhecível.

A prova é a mesma que a do Teorema 5.16, exceto que M e N são reconhecedores ao invés de decisores.

Corolário 5.23

Se $A \leq_m B$ e A não é Turing-reconhecível, então B não é Turing-reconhecível.

Em uma aplicação típica desse corolário, fazemos A ser $\overline{A_{\text{MT}}}$, o complemento de A_{MT} . Sabemos que $\overline{A_{\text{MT}}}$ não é Turing-reconhecível do Corolário 4.17. A definição de redutibilidade por mapeamento implica que $A \leq_m B$ significa o mesmo que $\overline{A} \leq_m \overline{B}$. Para provar que B não é reconhecível podemos mostrar que $A_{\text{MT}} \leq_m \overline{B}$. Podemos também usar redutibilidade por mapeamento para mostrar que certos problemas não são nem Turing-reconhecíveis nem co-Turing-reconhecíveis como no teorema seguinte.

Teorema 5.24

EQ_{MT} nem é Turing-reconhecível nem co-Turing-reconhecível.

Prova. Primeiro mostramos que EQ_{MT} não é Turing-reconhecível. Fazemos isso mostrando que A_{MT} é redutível a $\overline{EQ_{\text{MT}}}$. A função de redução f funciona da seguinte forma.

F = “Sobre a entrada $\langle M, w \rangle$ onde M é uma MT e w uma cadeia:

1. Construa as seguintes máquinas de Turing M_1 e M_2 .

M_1 = “Sobre qualquer entrada:

1. *Rejeite.*”

M_2 = “Sobre qualquer entrada:

1. Rode M sobre w . Se ela aceita, *aceite.*”

2. Dê como saída $\langle M_1, M_2 \rangle$.”

Aqui, M_1 não aceita nada. Se M aceita w , M_2 aceita tudo, e portanto as duas máquinas não são equivalentes. Reciprocamente, se M não aceita w , M_2 não aceita nada, e elas são equivalentes. Por conseguinte f reduz A_{MT} a $\overline{EQ_{\text{MT}}}$, como desejado.

Para mostrar que $\overline{EQ_{\text{MT}}}$ não é Turing-reconhecível damos uma redução de A_{MT} para o complemento de $\overline{EQ_{\text{MT}}}$, a saber, EQ_{MT} . Logo, mostramos que $A_{\text{MT}} \leq_m EQ_{\text{MT}}$. A seguinte MT G computa a função redutora g .

G = “A entrada é $\langle M, w \rangle$ onde M é uma MT e w uma cadeia.

1. Construa as seguintes máquinas de Turing M_1 e M_2 : M_1 = “Sobre qualquer entrada:

1. *Aceite.*”

M_2 = “Sobre qualquer entrada:

1. Rode M sobre w .

2. Se ela aceita, *aceite.*”

2. Dê como saída $\langle M_1, M_2 \rangle$.”

A única diferença entre f e g é na máquina M_1 . Em f , a máquina M_1 sempre rejeita, enquanto que em g ela sempre aceita. Em ambas f e g , M aceita w se e somente se M_2 sempre aceita. Em g , M aceita w se e somente se M_1 e M_2 são equivalentes. É por isso que g é uma redução de A_{MT} para EQ_{MT} .

.....

.....

Exercícios

5.1 Mostre que EQ_{GLC} é indecidível.

5.2 Mostre que EQ_{GLC} é co-Turing-reconhecível.

5.3 Encontre um emparelhamento na seguinte instância do PCP.

$$\left\{ \left[\frac{ab}{abab} \right], \left[\frac{b}{a} \right], \left[\frac{aba}{b} \right], \left[\frac{aa}{a} \right] \right\}$$

5.4 Se $A \leq_m B$ e B é uma linguagem regular, isso implica que A seja uma linguagem regular? Por que ou por que não?

5.5 Mostre que A_{MT} não é redutível por mapeamento a V_{MT} .

- 5.6** Mostre que \leq_m é uma relação transitiva.
- 5.7** Mostre que se A é Turing-reconhecível e $A \leq_m \overline{A}$, então A é decidível.
- 5.8** Na prova do Teorema 5.11 modificamos a máquina de Turing M de modo que ela nunca tenta mover sua cabeça para fora da extremidade esquerda da fita. Suponha que não fizéssemos essa modificação em M . Como teríamos que modificar a construção do PCP para lidar com esse caso?

Problemas

- 5.9** Mostre que todos os problemas Turing-reconhecíveis reduzem por mapeamento a A_{MT} .
- 5.10** Seja $J = \{w \mid w = 0x \text{ para algum } x \in A_{\text{MT}} \text{ ou } w = 1y \text{ para algum } y \in \overline{A_{\text{MT}}}\}$. Mostre que nem J nem \overline{J} é Turing-reconhecível.
- 5.11** Dê um exemplo de uma linguagem indecidível B , onde $B \leq_m \overline{B}$.
- 5.12** Seja $S = \{\langle M \rangle \mid M \text{ é uma MT que aceita } w^R \text{ sempre que ela aceita } w\}$. Mostre que S é indecidível.
- 5.13** Um *estado inútil* em uma máquina de Turing é aquele no qual a máquina nunca entra sobre nenhuma entrada. Considere o problema de se testar se uma máquina de Turing tem quaisquer estados inúteis. Formule esse problema como uma linguagem e mostre que ele é indecidível.
- 5.14** Considere o problema de se testar se uma máquina de Turing sobre uma entrada w em algum momento move sua cabeça para a esquerda quando sua cabeça está sobre a célula mais à esquerda. Formule esse problema como uma linguagem e mostre que ele é indecidível.
- 5.15** Considere o problema de se testar se uma máquina de Turing sobre uma entrada w em algum momento move sua cabeça para a esquerda em algum ponto durante sua computação sobre w . Formule esse problema como uma linguagem e mostre que ele é decidível.
- 5.16** Considere o problema de se testar se uma máquina de Turing de duas-fitas em algum momento escreve um símbolo não-branco na sua segunda fita. Formule esse problema como uma linguagem e mostre que ele é indecidível.
- 5.17** Mostre que o PCP é decidível sobre um alfabeto unário, ou seja, sobre o alfabeto $\Sigma = \{1\}$.
- 5.18** Mostre que o PCP é indecidível sobre um alfabeto binário, ou seja, sobre o alfabeto $\Sigma = \{0, 1\}$.
- 5.19** Seja $AMBIG_{\text{GLC}} = \{\langle G \rangle \mid G \text{ é uma GLC ambígua}\}$. Mostre que $AMBIG_{\text{GLC}}$ é indecidível. (Dica: Use uma redução do PCP. Dê uma instância

$$P = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\},$$

do PCP, construa uma GLC G com as regras

$$\begin{aligned} S &\rightarrow T \mid B \\ T &\rightarrow t_1 T a_1 \mid \dots \mid t_k T a_k \mid t_1 a_1 \mid \dots \mid t_k a_k \\ B &\rightarrow b_1 B a_1 \mid \dots \mid b_k B a_k \mid b_1 a_1 \mid \dots \mid b_k a_k, \end{aligned}$$

onde a_1, \dots, a_k são novos símbolos terminais. Prove que essa redução funciona.)

- 5.20** Defina um *autômato finito de duas-cabeças* (2AFD) como sendo um autômato finito determinístico que duas cabeças de somente-leitura, bidirecionais, que começam na extremidade esquerda da fita de entrada e podem ser controladas independentemente para mover em qualquer direção. A fita de um 2AFD é finita e é justamente longa o suficiente para conter a entrada mais duas células de fita em branco, uma na extremidade esquerda e outra na extremidade direita, que servem como delimitadores. Um 2AFD aceita sua entrada entrando num especial de parada. Por exemplo, um 2AFD pode reconhecer a linguagem $\{a^n b^n c^n \mid n \geq 0\}$.
- Seja $A_{2\text{AFD}} = \{\langle M, x \rangle \mid M \text{ é um 2AFD e } M \text{ aceita } x\}$. Mostre que $A_{2\text{AFD}}$ é decidível.
 - Seja $V_{2\text{AFD}} = \{\langle M, x \rangle \mid M \text{ é um 2AFD e } L(M) = \emptyset\}$. Mostre que $V_{2\text{AFD}}$ não é decidível.

- 5.21** Um *autômato finito bidimensional* (AFD-2DIM) é definido da seguinte forma. A entrada é um retângulo $m \times n$, para qualquer $m, n \geq 2$. Os quadrados ao longo da fronteira do retângulo contêm o símbolo # e os quadrados internos contêm símbolos sobre o alfabeto de entrada Σ . A função de transição é um mapeamento $Q \times \Sigma \rightarrow Q \times \{E, D, C, B\}$ para indicar o próximo estado e a nova posição da cabeça (Esquerda, Direita, Cima, Baixo). A máquina aceita quando ela entra em um dos estados de aceitação designados. Ela rejeita se ela tenta mover para fora do retângulo de entrada ou se ela nunca pára. Duas dessas máquinas são equivalentes se elas aceitam os mesmos retângulos. Considere o problema de se testar se duas dessas máquinas são equivalentes. Formule esse problema como uma linguagem, e mostre que ele é indecidível.

- *5.22 Teorema de Rice.** Seja P um problema qualquer sobre máquinas de Turing que satisfaz as duas propriedades abaixo. Como de costume expressamos P como uma linguagem.

- Para quaisquer MT's M_1 e M_2 , onde $L(M_1) = L(M_2)$, temos $\langle M_1 \rangle \in P$ se e só se $\langle M_2 \rangle \in P$. Em outras palavras, a pertinência de uma MT M em P depende somente da linguagem de M .
- Existem MT's M_1 e M_2 , onde $\langle M_1 \rangle \in P$ e $\langle M_2 \rangle \notin P$. Em outras palavras, P é não-trivial—ela se verifica para algumas, mas não todas, MT's.

Mostre que P é indecidível.

- 5.23** Mostre que ambas as condições no Problema 5.22 são necessárias para provar que P é indecidível.

Capítulo 6

Tópicos Avançados em Teoria da Computabilidade

- 6.1 O teorema da recursão**
- 6.2 Decidibilidade de teorias lógicas**
- 6.3 Redutibilidade de Turing**
- 6.4 Uma definição de informação**

Parte Três: Teoria da Complexidade

Capítulo 7

Complexidade de Tempo

Mesmo quando um problema é decidível e portanto computacionalmente solúvel em princípio, ele pode não ser solúvel na prática se a solução requer uma quantidade desordenada de tempo ou memória. Nesta parte final do livro introduzimos a teoria da complexidade computacional—uma investigação do tempo, memória, ou outros recursos requeridos para resolver problemas computacionais. Começamos com tempo.

Nosso objetivo neste capítulo é apresentar o básico da teoria da complexidade de tempo. Primeiro introduzimos uma maneira de medir o tempo usado para resolver um problema. Então mostramos como classificar problemas conforme a quantidade de tempo requerida. Depois disso discutimos a possibilidade de que certos problemas decidíveis requerem quantidades enormes de tempo e como determinar quando você se defronta com tal problema.

7.1 Medindo complexidade

Vamos começar com um exemplo. Tome a linguagem $A = \{0^k 1^k \mid k \geq 0\}$. Obviamente A é uma linguagem decidível. Quanto tempo uma máquina de uma única fita necessita para decidir A ? Examinamos a seguinte MT de uma única fita M_1 para A . Damos a descrição da máquina de Turing em baixo nível, incluindo o próprio movimento da cabeça sobre a fita, de modo que possamos contar o número de passos que M_1 usa quando ela roda.

M_1 = “Sobre a cadeia de entrada w :

1. Faça uma varredura na fita e *rejeite* se um 0 for encontrado à direita de um 1.
2. Repita o seguinte passo se ambos 0's e 1's permanecem na fita.
 3. Faça uma varredura na fita, riscando um único 0 e um único 1.
 4. Se 0's ainda permanecem após todos os 1's tenham sido riscados, ou se 1's ainda permanecem após todos os 0's tenham sido riscados, *rejeite*. Caso contrário, se nem 0's nem 1's permanecem na fita, *aceite*.”

Analisamos o algoritmo para a máquina de Turing M_1 que decide A para determinar quanto tempo ele usa.

O número de passos que um algoritmo usa sobre uma entrada específica pode depender de vários parâmetros. Por exemplo, se a entrada é um grafo, o número de passos pode depender do número de nós, do número de arestas, e do grau máximo do grafo, ou alguma combinação desses e/ou outros fatores. Por simplicidade computamos o tempo de execução de um algoritmo puramente como uma função do comprimento da cadeia que representa a entrada e não consideramos qualquer outro parâmetro. Na **análise do pior-caso**, a forma que consideramos aqui, consideramos o maior tempo de execução de todas as entradas de um comprimento específico. Na **análise do caso-médio** consideramos a média de todos os tempos de execução de entradas de um comprimento específico.

Definição 7.1

Seja M uma máquina de Turing determinística que pára sobre todas as entradas. O **tempo de execução** ou **complexidade de tempo** de M é a função $f : \mathcal{N} \rightarrow \mathcal{N}$, onde $f(n)$ é o número máximo de passos que M usa sobre qualquer entrada de comprimento n . Se $f(n)$ é o tempo de execução de M , dizemos que M roda em tempo $f(n)$ e que M é uma máquina de Turing de tempo $f(n)$.

Notação O -grande e o -pequeno

Devido ao fato de que o tempo exato de execução de um algoritmo frequentemente é uma expressão complexa, usualmente apenas o estimamos. Em uma forma conveniente de estimativa, chamada **análise assintótica**, buscamos entender o tempo de execução do algoritmo quando ele é rodado sobre entradas grandes. Fazemos isso considerando somente o termo de mais alta ordem da expressão para o tempo de execução do algoritmo, desconsiderando ambos os coeficientes daquele termo e quaisquer termos de mais baixa ordem, porque o termo de mais alta ordem domina os outros termos sobre entradas grandes.

Por exemplo, a função $f(n) = 6n^3 + 2n^2 + 20n + 45$ tem quatro termos, e o termo de mais alta ordem é $6n^3$. Desconsiderando o coeficiente 6, dizemos que f é assintoticamente no máximo n^3 . A **notação assintótica** ou **notação O -grande** para descrever esse relacionamento é $f(n) = O(n^3)$. Formalizamos essa noção na definição a seguir. Seja \mathcal{R}^+ o conjunto dos números reais maiores que 0.

Definição 7.2

Sejam f e g duas funções $f, g : \mathcal{N} \rightarrow \mathcal{R}^+$. Vamos dizer que $f(n) = O(g(n))$ se existem inteiros positivos c e n_0 existem tais que para todo inteiro $n \geq n_0$

$$f(n) \leq c g(n)$$

Quando $f(n) = O(g(n))$ dizemos que $g(n)$ é um **limitante superior** para $f(n)$, ou mais precisamente, que $g(n)$ é um **limitante superior assintótico** para $f(n)$, para enfatizar que estamos suprimindo fatores constantes.

Intuitivamente, $f(n) = O(g(n))$ significa que f é menor ou igual a g se desconsiderarmos diferenças até um fator constante. Você pode pensar em O como representando uma constante suprimida. Na prática, a maioria das funções f que você provavelmente vai encontrar têm um termo de mais alta ordem h óbvio. Nesse caso escrevemos $f(n) = O(g(n))$, onde g é h sem seu coeficiente.

Exemplo 7.3

Seja $f_1(n)$ a função $5n^3 + 2n^2 + 22n + 6$. Então, selecionando o termo de mais alta ordem $5n^3$ e desconsiderando o coeficiente 5 dá $f_1(n) = O(n^3)$.

Vamos verificar que esse resultado satisfaz a definição formal. Fazemos isso tornando c o valor 6 e n_0 o valor 10. Então, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ para todo $n \geq 10$.

Ademais, $f_1(n) = O(n^4)$ porque n^4 é maior que n^3 e portanto ainda é um limite superior assintótico sobre f_1 .

Entretanto, $f_1(n)$ não é $O(n^2)$. Independentemente de valores que atribuamos a c e n_0 , a definição permanece não-satisfeta nesse caso.

Exemplo 7.4

A notação O -grande interage com logaritmos de uma maneira particular. Usualmente quando utilizamos logaritmos temos que especificar a base, como em $x = \log_2 n$. A base 2 aqui indica que essa igualdade é equivalente à igualdade $2^x = n$. Mudar o valor da base b muda o valor de $\log_b n$ de um fator constante, devido à identidade $\log_b n = \log_2 n / \log_2 b$. Por conseguinte, quando escrevemos $f(n) = O(\log n)$, especificar a base não é mais necessário porque estamos suprimindo fatores constantes de qualquer forma.

Seja $f_2(n)$ a função $3n \log_2 n + 5n \log_2 \log_2 n + 2$. Nesse caso temos $f_2(n) = O(n \log n)$ porque $\log n$ domina $\log \log n$.

A notação O -grande também aparece em expressões aritméticas tais como a expressão $f(n) = O(n^2) + O(n)$. Nesse caso cada ocorrência do símbolo O representa uma constante suprimida diferente. Devido ao fato de que o termo $O(n^2)$ domina o termo $O(n)$, essa expressão é equivalente a $f(n) = O(n^2)$. Quando o símbolo O ocorre em um expoente como na expressão $f(n) = 2^{O(n)}$, a mesma idéia se aplica. Essa expressão representa um limite superior de 2^{cn} para alguma constante c .

A expressão $f(n) = 2^{O(\log n)}$ ocorre em algumas análises. Usando a identidade $n = 2^{\log_2 n}$ e por conseguinte que $n^c = 2^{c \log_2 n}$, vemos que $2^{O(\log n)}$ representa um limite superior de n^c para alguma constante c . A expressão $n^{O(1)}$ representa o mesmo limite de uma maneira diferente, porque a expressão $O(1)$ representa um valor que nunca é mais que uma constante fixa.

Frequentemente derivamos limitantes da forma n^c para c maior que 0. Tais limitantes são chamados **limitantes polinomiais**. Limitantes da forma $2^{(n^\delta)}$ são chamados **limitantes exponenciais** quando δ é um número real maior que 0.

A notação O -grande tem uma companheira chamada **notação o-pequeno**. A notação O -grande dá uma maneira de se dizer que uma função é assintoticamente **não mais que uma outra**. Para dizer que uma função é assintoticamente *menor que* uma outra usamos a notação *o-pequeno*. A diferença entre as notações O -grande e *o-pequeno* é análoga à diferença entre \leq e $<$.

Definição 7.5

Sejam f e g duas funções $f, g : \mathcal{N} \rightarrow \mathcal{R}^+$ se

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

Em outras palavras, $f(n) = o(g(n))$ significa que, para qualquer número real $c > 0$, um número n_0 existe, onde $f(n) < c g(n)$ para todo $n > n_0$.

Exemplo 7.6

Os seguintes são fáceis de verificar.

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.

Entretanto, $f(n)$ nunca é $o(f(n))$.

Analisando algoritmos

Vamos analisar o algoritmo que demos para a linguagem $A = \{0^k 1^k \mid k \geq 0\}$. Repetimo-lo aqui por conveniência.

M_1 = “Sobre a cadeia de entrada w :

1. Faç uma varredura na fita e *rejeite* se um 0 for encontrado à direita de um 1.
2. Repita o seguinte passo se ambos 0's e 1's permanecem na fita.
3. Faça uma varredura na fita, riscando um único 0 e um único 1.
4. Se 0's ainda permanecem após todos os 1's tenham sido riscados, ou se 1's ainda permanecem após todos os 0's tenham sido riscados, *rejeite*. Caso contrário, se nem 0's nem 1's permanecem na fita, *aceite*.”

Para analisar M_1 consideramos cada um dos seus três estágios separadamente. No estágio 1, a máquina faz uma varredura na fita para verificar se a entrada é da forma $0^* 1^*$. Realizar essa varredura usa n passos. Reposicionar a cabeça na extremidade esquerda da fita usa outros n passos. Portanto o total usado nesse estágio é $2n$ passos. Em notação O -grande dizemos que esse estágio usa $O(n)$ passos. Note que não mencionamos o reposicionamento da cabeça da fita na descrição da máquina. Usar notação assintótica nos permite omitir detalhes da descrição da máquina que afetam o tempo de execução por no máximo um fator constante.

Nos estágios 2 e 3, a máquina repetidamente faz uma varredura na fita e risca um 0 e um 1 em cada varredura. Cada varredura usa $O(n)$ passos. Como cada varredura risca dois símbolos, no máximo $n/2$ varreduras pode ocorrer. Portanto o tempo total tomado pelos estágios 2 e 3 é $(n/2)O(n) = O(n^2)$ passos.

No estágio 4 a máquina faz uma única varredura para decidir se aceita ou rejeita. O tempo tomado nesse estágio é no máximo $O(n)$.

Por conseguinte o tempo total de M_1 sobre qualquer entrada de comprimento n é $O(n) + O(n^2) + O(n)$, ou $O(n^2)$. Em outras palavras, seu tempo de execução é $O(n^2)$, o que completa a análise de tempo dessa máquina.

Vamos estabelecer alguma notação para classificar linguagens conforme seus requisitos de tempo.

Definição 7.7

Seja $t : \mathcal{N} \rightarrow \mathcal{N}$ uma função. Defina a **classe de complexidade de tempo** $\text{TIME}(t(n))$, como sendo

$$\text{TIME}(t(n)) = \{L \mid L \text{ é uma linguagem decidida por uma MT de tempo } O(t(n))\}.$$

Lembremo-nos que a linguagem $A = \{0^k 1^k \mid k \geq 0\}$. A análise precedente mostra que $A \in \text{TIME}(n^2)$ porque M_1 decide A em tempo $O(n^2)$ e $\text{TIME}(n^2)$ contém todas as linguagens que podem ser decididas em tempo $O(n^2)$.

Existe uma máquina que decide A assintoticamente mais rapidamente? Em outras palavras, A pertence a $\text{TIME}(t(n))$ para $t(n) = o(n^2)$? Podemos melhorar o tempo de execução riscando dois 0's e dois 1's em cada varredura ao invés de somente um pois fazer isso corta o número de varreduras pela metade. Mas isso melhora o tempo de execução somente por um fator de 2 e não afeta o tempo de execução assintótico. A máquina a seguir, M_2 , usa um método diferente para decidir A assintoticamente mais rápido. Ela mostra que $A \in \text{TIME}(n \log n)$.

M_2 = “Sobre a cadeia de entrada w :

1. Faça uma varredura na fita e *rejeite* se um 0 for encontrado à direita de um 1.
2. Repita os seguintes passos enquanto alguns 0's e alguns 1's permaneçam na fita.
3. Faça uma varredura na fita, verificando se o número total de 0's e 1's remanescentes é par ou ímpar. Se for ímpar, *rejeite*.
4. Faça novamente uma varredura na fita, riscando um 0 sim e outro não começando com o primeiro 0, e então riscando um 1 sim e outro não começando com o primeiro 1.
5. Se nenhum 0 e nenhum 1 permanece na fita, *aceite*. Caso contrário, *rejeite*.”

Antes de analisar M_2 , vamos verificar que ela realmente decide A . Em cada varredura realizada no estágio 4, o número total de 0's remanescentes é cortado pela metade e qualquer resto é descartado. Por conseguinte, se começássemos com 13 0's, após o estágio 4 ser executado uma única vez somente 6 0's permanecem. Após execuções subsequentes desse estágio, 3, então 1, e então 0 permanecem. Esse estágio tem o mesmo efeito sobre o número de 1's.

Agora examinamos a paridade par/ímpar do número de 0's e o número de 1's em cada execução do estágio 3. Considere novamente começar com 13 0's e 13 1's. A primeira execução do estágio 3 encontra um número ímpar de 0's (porque 13 é um número ímpar) e um número ímpar de 1's. Em execuções subsequentes um número par (6) ocorre, então um número ímpar (3), e um número ímpar (1). Não executamos esse estágio sobre 0 0's ou 0 1's por causa da condição no laço de repetição especificado no estágio 2. Para a seqüência de paridades encontradas (ímpar, par, ímpar, ímpar) se substituirmos as pares por 0's e as ímpares por 1's e então reverter a seqüência, obtemos 1101, a representação binária de 13, ou o número de 0's e 1's no início. A seqüência de paridades sempre dá o reverso da representação binária.

Quando o estágio 3 verifica para determinar que o número total de 0's e 1's remanescente é par, ele na verdade está verificando a concordância da paridade de 0's com a paridade de 1's. Se todas as paridades concordam, a representação binária dos números de 0's e 1's concordam, e portanto os dois números são iguais.

Para analisar o tempo de execução de M_2 , primeiro observamos que todo estágio toma o tempo $O(n)$. Então determinamos o número de vezes que cada um é executado. Estágios 1 e 5 são executados uma vez, levando um total de $O(n)$. O estágio 4 risca pelo menos metade dos 0's e 1's a cada vez que ele é executado, portanto no máximo $1 + \log_2 n$ iterações ocorrem antes que todos estejam riscados. Por conseguinte o tempo total dos estágios 2, 3, e 4 $(1 + \log_2 n)O(n)$, ou $O(n \log n)$. O tempo de execução de M_2 é $O(n) + O(n \log n) = O(n \log n)$.

Antes mostramos que $A \in \text{TIME}(n^2)$, mas agora temos um limite melhor, a saber, $A \in \text{TIME}(n \log n)$. Esse resultado não pode ser melhorado ainda mais em máquinas de Turing de uma única fita. Na verdade, qualquer linguagem que pode ser decidida em tempo $O(n \log n)$ numa máquina de Turing de uma única fita é regular, embora não provaremos esse resultado.

Podemos decidir a linguagem A em tempo $O(n)$ (também chamado **tempo linear**) se a máquina de Turing tem uma segunda fita. A MT M_3 a seguir decide A em tempo linear.

M_3 = “Sobre a cadeia de entrada w :

1. Faça uma varredura na fita e *rejeite* se um 0 for encontrado à direita de um 1.
2. Faça uma varredura nos 0's na Fita 1 até o primeiro 1. Ao mesmo tempo copie os 0's para a Fita 2.
3. Faça uma varredura nos 1's na Fita 1 até o final da entrada. Para cada 1 lido na Fita 1, risque um 0 na Fita 2. Se todos os 0's estão riscados antes que todos os 1's sejam lidos, *rejeite*.
4. Se todos os 0's foram agora riscados, *aceite*. Se quaisquer 0's permanecem, *rejeite*.”

A máquina M_3 opera diferentemente das máquinas anteriores para A . Ela simplesmente copia os 0's para sua segunda fita e então os emparelha com os 1's.

Essa máquina é simples de analisar. Cada um dos quatro estágios obviamente usa $O(n)$ passos, portanto o tempo total de execução é $O(n)$ e por conseguinte linear. Note que esse tempo de execução é o melhor possível porque n passos são necessários só para ler a entrada.

Vamos resumir o que mostramos sobre a complexidade de tempo de A . Produzimos uma MT de uma única fita M_1 que decide A em tempo $O(n^2)$ e uma MT de uma única fita M_2 que decide A em tempo $O(n \log n)$. Afirmamos (sem prova) que nenhuma MT de uma única fita pode fazê-lo mais rapidamente. Então exibimos uma MT de duas-fitas M_3 que decide A em tempo $O(n)$. Logo a complexidade de tempo de A sobre uma MT de uma única fita é $O(n \log n)$ e sobre uma MT de duas-fitas é $O(n)$. Note que a complexidade de A depende do modelo de computação escolhido.

Esta discussão levanta uma importante diferença entre a teoria da complexidade e a teoria da computabilidade. Na teoria da computabilidade, a tese de Church–Turing implica que todos os modelos razoáveis de computação são equivalentes, ou seja, eles todos decidem a mesma classe de linguagens. Na teoria da complexidade, a escolha do modelo afeta a complexidade de tempo de linguagens. Linguagens que são decidíveis em, digamos, tempo linear em um modelo não são necessariamente decidíveis em tempo linear em um outro.

Na teoria da complexidade, desejamos classificar problemas computacionais conforme a quantidade de tempo requerido para a solução. Mas com qual modelo medimos

o tempo? A mesma linguagem pode ter diferentes requisitos de tempo em diferentes modelos.

Felizmente, requisitos de tempo não diferem largamente para modelos determinísticos típicos. Portanto, se nosso sistema de classificação não é muito sensível a diferenças relativamente pequenas em complexidade, o modelo determinístico escolhido não é crucial. Discutimos essa idéia em mais detalhes nas próximas seções.

Relacionamentos de complexidade entre modelos

Aqui examinamos como a escolha do modelo computacional pode afetar a complexidade de tempo de linguagens. Consideramos três modelos: a máquina de Turing com uma única fita; a máquina de Turing multi-fita; e a máquina de Turing não-determinística.

Teorema 7.8

Seja $t(n)$ uma função, onde $t(n) \geq n$. Então toda máquina de Turing multi-fita de tempo $t(n)$ tem uma máquina de Turing de uma única fita equivalente de tempo $O(t^2(n))$.

Idéia da prova. A idéia por trás da prova desse teorema é bastante simples. Lembre-se que no Teorema 3.8 mostramos como converter qualquer MT multi-fita numa MT de uma única fita que a simula. Agora analisamos aquela simulação para determinar quanto tempo adicional ela requer. Mostramos que simular cada passo da máquina multi-fita usa no máximo $O(t(n))$ passos na máquina de uma única fita. Logo, o tempo total usado é $O(t^2(n))$ passos.

Prova. Seja M a MT de k -fitas que roda em tempo $t(n)$. Construimos uma MT de uma única fita S que roda em tempo $O(t^2(n))$.

A máquina S opera simulando M , como descrito no Teorema 3.8. Para revisar aquela simulação, relembremo-nos que S usa sua única fita para representar o conteúdo sobre todas as k fitas de M . As fitas são armazenadas consecutivamente, com as posições das cabeças de M marcadas sobre as células apropriadas.

Inicialmente, S põe sua fita no formato que representa todas as fitas de M e então simula os passos de M . Para simular um passo, S faz uma varredura em toda a informação armazenada em sua fita para determinar os símbolos sob as cabeças de M . Então S faz uma nova varredura sobre sua fita para atualizar o conteúdo da fita e as posições das cabeças. Se uma das cabeças de M move para a direita para uma parte previamente não lida de sua fita, S tem que aumentar a quantidade de espaço alocado para sua fita. Ela faz isso deslocando uma parte de sua própria fita uma célula para a direita.

Agora analisamos essa simulação. Para cada passo de M , a máquina S faz duas varreduras sobre a parte ativa de sua fita. A primeira obtém a informação necessária para determinar o próximo movimento e a segunda o realiza. O comprimento da parte ativa da fita de S determina quanto tempo S leva para fazer a varredura, portanto temos que determinar um limite superior sobre esse comprimento. Para fazer isso tomamos a soma dos comprimentos das partes ativas das k fitas de M . Cada uma dessas partes ativas tem comprimento no máximo $t(n)$ pois M usa $t(n)$ células de fita em $t(n)$ passos se a cabeça move para a direita a cada passo e até menos se uma cabeça em

algum momento move para a esquerda. Por conseguinte uma varredura da parte ativa da fita de S usa $O(t(n))$ passos.

Para simular cada um dos passos de M , S realiza duas varreduras e possivelmente até k deslocamentos para a direita. Cada um usa o tempo $O(t(n))$, portanto o tempo total para S simular um dos passos de M é $O(t(n))$.

Agora limitamos o tempo total usado pela simulação. O estágio inicial, onde S põe sua fita no formato apropriado, usa $O(n)$ passos. Depois disso, S simula cada um dos $t(n)$ passos de M , usando $O(t(n))$ passos, portanto essa parte da simulação usa $t(n) \times O(t(n)) = O(t^2(n))$ passos. Por conseguinte a simulação inteira de M usa $O(n) + O(t^2(n))$ passos.

Assumimos que $t(n) \geq n$ (uma suposição razoável porque M poderia nem sequer ler toda a entrada em menos tempo). Por conseguinte o tempo de execução de S é $O(t^2(n))$ e a prova está completa.

A seguir, consideramos o teorema análogo para máquinas de Turing não-determinísticas de uma única fita. Mostramos que qualquer linguagem que é decidível em tal máquina é decidível em uma máquina de Turing determinística de uma única fita que requer significativamente mais tempo. Antes de fazer isso, temos que definir o tempo de execução de uma máquina de Turing não-determinística. Relembremo-nos de que uma máquina de Turing não-determinística é um decisor se todos os ramos de sua computação param sobre todas as entradas.

Definição 7.9

Seja N uma máquina de Turing não-determinística que é um decisor. O **tempo de execução** de N é a função $f : \mathcal{N} \rightarrow \mathcal{N}$, onde $f(n)$ é o número máximo de passos que N usa em qualquer ramo de sua computação sobre qualquer entrada de comprimento n , como mostrado na Figura 7.1.

Figura 7.1: Medindo tempo determinístico e não-determinístico

A definição de tempo de execução de uma máquina de Turing não-determinística não pretende corresponder a nenhum dispositivo de computação do mundo-real. Ao contrário, ela é uma definição matemática útil que assiste na caracterização da complexidade de uma importante classe de problemas computacionais, como demonstraremos em breve.

Teorema 7.10

Seja $t(n)$ uma função, onde $t(n) \geq n$. Então toda máquina de Turing não-determinística de uma única fita de tempo $t(n)$ tem uma máquina de Turing determinística de uma única fita equivalente de tempo $2^{O(t(n))}$.

Prova. Seja N uma MT não-determinística rodando em tempo $t(n)$. Construimos uma MT determinística D que simula N como na prova do Teorema 3.10 por meio de uma busca na árvore de computação não-determinística de N . Agora analisamos aquela simulação.

Sobre uma entrada de comprimento n , todo ramo da árvore de computação não-determinística de N tem um comprimento no máximo $t(n)$. Todo nó na árvore pode ter no máximo b filhos, onde b é o número máximo de escolhas legítimas dadas pela função de transição de N . Por conseguinte o número total de folhas na árvore é no máximo $b^{t(n)}$.

A simulação procede explorando essa árvore por largura. Em outras palavras, ela visita todos os nós de profundidade d antes de ir adiante para qualquer nó de profundidade $d + 1$. O algoritmo dado na prova do Teorema 3.10 inefficientemente começa na raiz e desce para um nó sempre que ele visita aquele nó, mas eliminar essa inefficiência não altera o enunciado do teorema corrente, portanto deixamo-lo como está. O número total de nós na árvore é menor que duas vezes o número máximo de folhas, portanto limitamo-lo por $O(b^{t(n)})$. O tempo para iniciar a partir da raiz e descer até um nó é $O(t(n))$. Por conseguinte o tempo de execução de D é $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

Como descrito no Teorema 3.10, a MT D tem três fitas. Convertê-la para uma MT de uma única fita no máximo eleva ao quadrado o tempo de execução, pelo Teorema 7.8. Por conseguinte o tempo de execução do simulador de uma única fita é $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$, e o teorema está provado.

.....

7.2 A classe P

Os Teoremas 7.8 e 7.10 ilustram uma importante distinção. Por um lado, provamos uma diferença no máximo quadrática ou *polinomial* entre a complexidade de tempo de problemas medidos sobre máquinas de Turing determinísticas de uma única fita e multi-fitas. Por outro lado, provamos uma diferença no máximo *exponencial* entre a complexidade de tempo de problemas medidos sobre máquinas de Turing determinísticas e não-determinísticas.

Tempo polinomial

Para nossos propósitos, diferenças polinomiais em tempo de execução são consideradas pequenas, enquanto que diferenças exponenciais são consideradas grandes. Vamos olhar por que fazer essa separação entre polinômios e exponenciais ao invés de entre algumas outras classes de funções.

Primeiro, note a diferença dramática entre a taxa de crescimento de polinômios que tipicamente ocorrem tal como n^3 e exponenciais que tipicamente ocorrem tal como 2^n . Por exemplo, suponha que n seja 1000, o tamanho de uma entrada razoável para um algoritmo. Nesse caso, n^3 é 1 bilhão, um número grande, porém, administrável, enquanto que 2^n é um número muito maior que o número de átomos no universo. Algoritmos de tempo polinomial são suficientemente rápidos para muitos propósitos, mas algoritmos de tempo exponencial raramente são úteis.

Algoritmos de tempo exponencial surgem quando resolvemos problemas por meio de busca através de um espaço de soluções, chamado **busca por força-bruta**. Por exemplo, uma maneira de fatorar um número em seus primos constituintes é buscar através de seus potenciais divisores. O tamanho do espaço de busca é exponencial, portanto essa busca usa tempo exponencial. Às vezes, busca por força-bruta pode ser evitada por meio de um entendimento mais profundo de um problema, que pode revelar um algoritmo de tempo polinomial de maior utilidade.

Todos os modelos computacionais determinísticos razoáveis são **polinomialmente equivalentes**. Ou seja, qualquer um deles pode simular um outro com apenas um acréscimo polinomial em tempo de execução. Quando dizemos que todos os modelos determinísticos razoáveis são polinomialmente equivalentes, não tentamos definir *razoável*. Entretanto, temos em mente uma noção suficientemente ampla para incluir modelos que se aproximam bastante dos tempos de execução sobre computadores reais. Por exemplo, o Teorema 7.8 mostra que os modelos de máquina de Turing determinística de uma única fita e multi-fita são polinomialmente equivalentes.

Daqui em diante nos concentramos em aspectos da teoria da complexidade de tempo que não são afetados por diferenças polinomiais em tempo de execução. Consideramos tais diferenças como sendo insignificantes e as ignoramos. Fazer isso nos permite desenvolver a teoria de uma maneira que não depende da escolha de um modelo específico de computação. Lembre-se, nosso objetivo é apresentar as propriedades fundamentais da *computação*, ao invés de propriedades de máquinas de Turing ou qualquer outro modelo especial.

Você pode achar que desconsiderar diferenças polinomiais em tempo de execução é absurdo. Programadores reais certamente se preocupam com tais diferenças e trabalham árduo só para fazer seus programas rodar duas vezes mais rápido. Entretanto, desconsideramos fatores constantes há um tempo atrás quando introduzimos notação assintótica. Agora propomos desconsiderar as diferenças polinomiais ainda maiores, tal como aquela entre tempo n e tempo n^3 .

Nossa decisão de desconsiderar diferenças polinomiais não implica que consideramos tais diferenças não-importantes. Ao contrário, certamente consideramos a diferença entre tempo n e tempo n^3 como sendo uma diferença importante. Mas algumas questões, tal como a polinomialidade ou não-polinomialidade do problema da fatoração, não dependem de diferenças polinomiais e são importantes também. Simplesmente escolhemos nos concentrar nesse tipo de questão aqui. Ignorar as árvores para ver a floresta não significa que uma é mais importante que a outra—isso simplesmente dá uma perspectiva diferente.

Agora chegamos a uma importante definição em teoria da complexidade.

Definição 7.11
 P é a classe de linguagens que são decidíveis em tempo polinomial sobre uma máquina de Turing determinística de uma única fita. Em outras palavras,

$$P = \bigcup_k \text{TIME}(n^k).$$

A classe P desempenha um papel central em nossa teoria e é importante porque

1. P é invariante para todos os modelos de computação que são polinomialmente equivalentes à máquina de Turing determinística de uma única fita, e
2. P corresponde aproximadamente à classe de problemas que são solúveis realisticamente em um computador.

Item 1 indica que P é uma classe matematicamente robusta. Ela não é afetada pelos particulares do modelo de computação que estamos usando.

Item 2 indica que P é relevante de um ponto de vista prático. Quando um problema está em P , temos um método de resolvê-lo que roda em tempo n^k para alguma constante k . Se esse tempo de execução é prático depende de k e da aplicação. É

claro que é improvável que um tempo de execução de n^{100} seja de algum uso prático. Não obstante, chamar o tempo polinomial de o limiar da solubilidade prática tem provado ser de utilidade. Uma vez que um algoritmo de tempo polinomial tenha sido encontrado para um problema que anteriormente parecia requerer tempo exponencial, alguma percepção chave sobre ele foi obtida, e reduções adicionais na sua complexidade usualmente seguem, frequentemente até o ponto de utilidade prática real.

Exemplos de problemas em P

Quando apresentamos um algoritmo de tempo polinomial damos uma descrição de alto-nível sem referência a características de um modelo computacional específico. Fazer isso evita detalhes tediosos de fitas e movimentos de cabeça. Precisamos seguir certas convenções ao descrever um algoritmo de modo que possamos analisá-lo com respeito a polinomialidade.

Descrevemos algoritmos com estágios numerados. A noção de um estágio de um algoritmo é análoga a um passo de uma máquina de Turing, embora obviamente, implementar um estágio de um algoritmo numa máquina de Turing, em geral, vai requerer muitos passos de máquina de Turing.

Quando analisamos um algoritmo para mostrar que ele roda em tempo polinomial, precisamos fazer duas coisas. Primeiro, temos que dar um limitante superior polinomial (usualmente em notação O -grande) sobre o número de estágios que o algoritmo usa quando ele roda sobre uma entrada de comprimento n . Então, temos que examinar os estágios individuais na descrição do algoritmo para assegurar que cada um pode ser implementado em tempo polinomial sobre um modelo determinístico razoável. Escolhemos os estágios quando descrevemos o algoritmo para tornar essa segunda parte da análise fácil de fazer. Quando ambas as tarefas tiverem sido completadas, podemos concluir que o algoritmo roda em tempo polinomial porque demonstramos que ele roda por um número polinomial de estágios, cada um dos quais pode ser feito em tempo polinomial, e a composição de polinômios é um polinômio.

Um ponto que requer atenção é o método de codificação usado para problemas. Continuamos a usar a notação $\langle \cdot \rangle$ para indicar uma codificação razoável de um ou mais objetos em uma cadeia, sem especificar qualquer método específico de codificação. Agora, um método razoável é aquele que permite codificação e decodificação polinomial de objetos em representações internas naturais ou em outras codificações razoáveis. Métodos de codificação familiares para grafos, autômatos, e semelhantes são todos razoáveis. Mas note que notação unária para codificar números (como no número 17 codificado pela cadeia unária 11111111111111) não é razoável porque ela é exponencialmente maior que codificações verdadeiramente razoáveis, tais como a notação na base k para qualquer $k \geq 2$.

Muitos problemas computacionais que você encontra neste capítulo contêm codificações de grafos. Uma codificação razoável de um grafo é uma lista de seus nós e arestas. Uma outra é a **matriz de adjacência**, onde a (i, j) -ésima entrada é 1 se existe uma aresta do nó i para o nó j e 0 se não existe. Quando analisamos algoritmos sobre grafos, o tempo de execução pode ser computado em termos do número de nós ao invés do tamanho da representação do grafo. Em representações razoáveis de grafos, o tamanho da representação é um polinômio no número de nós. Por conseguinte, se analisarmos um algoritmo e mostrarmos que seu tempo de execução é polinomial (ou exponencial) no número de nós, sabemos que ele é polinomial (ou exponencial) no tamanho da entrada.

O primeiro problema concerne grafos direcionados. Um grafo direcionado G contém nós s e t , como mostrado na Figura 7.2. O problema $CAMINHO$ é determinar se um caminho direcionado existe de s para t . Seja

$$CAMINHO = \{\langle G, s, t \rangle \mid G \text{ é um grafo direcionado que tem um caminho direcionado de } s \text{ para } t\}.$$

Figura 7.2: O problema $CAMINHO$: Existe um caminho de s para t ?

Teorema 7.12
 $CAMINHO \in P$.

Idéia da prova. Provamos esse teorema apresentando um algoritmo de tempo polinomial que decide $CAMINHO$. Antes de descrever esse algoritmo, vamos observar que um algoritmo de força-bruta para esse problema não é suficientemente rápido.

Um algoritmo de força-bruta para $CAMINHO$ procede examinando todos os caminhos em potencial em G e determinando se qualquer deles é um caminho direcionado de s para t . Um caminho em potencial é uma sequência de nós em G . (Se algum caminho direcionado existe de s para t , um tendo um comprimento de no máximo m existe porque repetir um nó nunca é necessário.) Mas o número de tais caminhos em potencial é m^m , que é exponencial no número de nós em G . Por conseguinte esse algoritmo de força-bruta usa tempo exponencial.

Para obter um algoritmo de tempo polinomial para $CAMINHO$ você tem que fazer algo que evite a força bruta. Uma maneira é usar um método de busca em grafo tal como busca em largura. Aqui, marcamos sucessivamente todos os nós em G que são atingíveis a partir de s por caminhos direcionados de comprimento 1, então 2, então 3, até m . Limitar o tempo de execução dessa estratégia por um polinômio é fácil.

Prova. Um algoritmo de tempo polinomial M para $CAMINHO$ opera da seguinte maneira:

M = “Sobre a entrada $\langle G, s, t \rangle$ onde G é um grafo direcionado com nós s e t :

1. Coloque uma marca sobre o nó s .
2. Repita os seguintes passos até que nenhum nó adicional esteja marcado.
3. Faça uma varredura nas arestas de G . Se uma aresta (a, b) for encontrada indo de um nó marcado a para um nó não marcado b , marque o nó b .
4. Se t estiver marcado, *aceite*. Caso contrário, *rejeite*.

Agora analisamos esse algoritmo para mostrar que ele roda em tempo polinomial. Obviamente, os estágios 1 e 4 são executados apenas uma vez. O estágio 3 roda no máximo m vezes porque cada vez exceto a última ele marca um nó adicional em G . Por conseguinte o número total de estágios usados é no máximo $1 + 1 + m$, dando um polinômio no tamanho de G .

Os estágios 1 e 4 de M são facilmente implementados em tempo polinomial sobre qualquer modelo determinístico razoável. O estágio 3 envolve uma varredura da

entrada e um teste se certos nós estão marcados, o que também é facilmente implementado em tempo polinomial. Portanto M é um algoritmo de tempo polinomial para $CAMINHO$.

Vamos nos voltar para um outro exemplo de um algoritmo de tempo polinomial. Vamos dizer que dois números são **primos entre si** se 1 é o maior inteiro que divide ambos. Por exemplo, 10 e 21 são primos entre si, muito embora nenhum dos dois seja um número primo em si próprio, enquanto que 10 e 22 não são primos entre si porque ambos são divisíveis por 2. Seja $PRIMENTSI$ o problema de se testar se dois números são primos entre si. Por conseguinte

$$PRIMENTSI = \{\langle x, y \rangle \mid x \text{ e } y \text{ são primos entre si}\}.$$

Teorema 7.13
 $PRIMENTSI \in P$.

Idéia da prova. Um algoritmo que resolve esse problema faz uma busca por todos os possíveis divisores de ambos os números e aceita se nenhum deles é maior que 1. Entretanto, a magnitude de um número representado em binário, ou na notação de qualquer outra base k para $k \geq 2$, é exponencial no comprimento de sua representação. Por conseguinte esse algoritmo de força-bruta busca através de um número exponencial de potenciais divisores e tem um tempo exponencial.

Ao invés disso, resolvemos esse problema com um procedimento numérico antigo, chamado de **algoritmo euclídeo**, para calcular o máximo divisor comum. O **máximo divisor comum** de dois números naturais x e y , escrito $\text{mdc}(x, y)$, é o maior inteiro que divide ambos x e y . Por exemplo, $\text{mdc}(18, 24) = 6$. Obviamente, x e y são primos entre si se $\text{mdc}(x, y) = 1$. Descrevemos o algoritmo euclídeo como algoritmo E na prova. Ele usa a função mod , onde $x \text{ mod } y$ é o resto da divisão de x por y .

Prova. O algoritmo euclídeo, E , é como segue.

E = “Sobre a entrada $\langle x, y \rangle$, onde x e y são números naturais em binário:

1. Repita até que $y = 0$.
2. Faça a atribuição $x \leftarrow x \text{ mod } y$.
3. Intercambie os conteúdos de x e y .
4. Dê como saída x .”

O algoritmo R resolve $PRIMENTSI$, usando E como uma subrotina.

R = “Sobre a entrada $\langle x, y \rangle$, onde x e y são números naturais em binário:

1. Rode E sobre $\langle x, y \rangle$.
2. Se o resultado for 1, *aceite*. Caso contrário, *rejeite*.

Claramente, se E roda corretamente em tempo polinomial, assim o faz também R e portanto precisamos apenas de analisar E por tempo e corretude. A corretude desse algoritmo é bem conhecida portanto não a discutiremos mais aqui.

Para analisar a complexidade de tempo de E , primeiro mostramos que toda execução do estágio 2 (exceto possivelmente o primeiro), corta o valor de x em pelo menos a metade. Após o estágio 2 ser executado, $x < y$ devido à natureza da função mod. Após o estágio 3, $x > y$ porque os dois tiveram seus conteúdos intercambiados. Por conseguinte, quando o estágio 2 é subsequentemente executado, $x > y$. Se $x/2 \geq y$, então $x \bmod y < y \leq x/2$ e x cai no mínimo pela metade. Se $x/2 < y$, então $x \bmod y = x - y < x/2$ e x cai no mínimo pela metade.

Os valores de x e y são intercambiados toda vez que o estágio 3 é executado, portanto cada um dos valores originais de x e y são reduzidos a pelo menos metade em passagens alternadas pelo laço. Por conseguinte o número máximo de vezes que os estágios 2 e 3 são executados é o mínimo entre $\log_2 x$ e $\log_2 y$. Esses logaritmos são proporcionais aos comprimentos das representações, dando o número de estágios executados como $O(n)$. Cada estágio de E usa somente tempo polinomial, portanto o tempo total de execução é polinomial.

O exemplo final de um algoritmo de tempo polinomial mostra que toda linguagem livre-do-contexto é decidível em tempo polinomial.

Teorema 7.14
Toda linguagem livre-do-contexto é um membro de P.

Idéia da prova. No Teorema 4.8 provamos que toda LLC é decidível. Para fazer isso demos um algoritmo para cada LLC que a decide. Se aquele algoritmo roda em tempo polinomial, o teorema corrente segue como um corolário. Vamos relembrar aquele algoritmo e descobrir se ele roda rapidamente o suficiente.

Seja L uma LLC gerada por uma GLC G que está na forma normal de Chomsky. Do Problema 2.19, qualquer derivação de uma cadeia w tem $2n - 1$ passos, onde n é o comprimento de w , devido ao fato de G está na forma normal de Chomsky. O decisor para L funciona tentando todas as possíveis derivações com $2n - 1$ passos quando sua entrada é uma cadeia de comprimento n . Se qualquer dessas for uma derivação de w , o decisor aceita; se não, ele rejeita.

Uma análise rápida desse algoritmo mostra que ele não roda em tempo polinomial. O número de derivações com k passos pode ser exponencial em k , portanto esse algoritmo pode requerer tempo exponencial.

Para obter um algoritmo de tempo polinomial introduzimos uma técnica poderosa chamada **programação dinâmica**. Essa técnica usa a acumulação de informação sobre subproblemas menores para resolver problemas maiores. Memorizamos a solução para qualquer subproblema de tal modo que precisamos resolvê-lo apenas uma vez. Fazemos isso montando uma tabela com todos os subproblemas e entrando com suas soluções sistematicamente à medida que as encontramos.

Neste caso, consideramos os subproblemas de se determinar se cada variável em G gera cada subcadeia de w . O algoritmo entra com a solução para esse subproblema numa tabela $n \times n$. Para $i \leq j$ a (i, j) -ésima entrada da tabela contém a coleção de variáveis que geram a subcadeia $w_i w_{i+1} \cdots w_j$. Para $i > j$ as entradas na tabela não são usadas.

O algoritmo preenche na tabela as entradas para cada subcadeia de w . Primeiro ele preenche nas entradas para as subcadeias de comprimento 1, aí então aquelas de comprimento 2, e assim por diante. Ele usa as entradas para comprimentos menores para ajudar a determinar as entradas para comprimentos maiores.

Por exemplo, suponha que o algoritmo já determinou quais variáveis gera todas as subcadeias até o comprimento k . Para determinar se uma variável A gera uma subcadeia específica de comprimento $k+1$ o algoritmo parte aquela subcadeia em dois pedaços não-vazios nas k maneiras possíveis. Para cada divisão, o algoritmo examina cada regra $A \rightarrow BC$ para determinar se B gera o primeiro pedaço e C gera o segundo pedaço, usando as entradas na tabela que já foram computadas. Se ambas B e C geram os respectivos pedaços, A gera a subcadeia e portanto é adicionada à entrada na tabela associada. O algoritmo começa o processo com as cadeias de comprimento 1 examinando a tabela para as regras $A \rightarrow b$.

Prova. O seguinte algoritmo D implementa a idéia da prova. Seja G uma GLC na forma normal de Chomsky L . Assuma que S seja a variável inicial. (Lembre-se de que a cadeia vazia é tratada especialmente numa gramática na forma normal de Chomsky. O algoritmo trata o caso especial no qual $w = \varepsilon$ no estágio 1.) Os comentários aparecem entre parênteses.

$D =$ Sobre a entrada $w = w_1 \cdots w_n$:

1. Se $w = \varepsilon$ e $S \rightarrow \varepsilon$ é uma regra, *aceite*. [[trata o caso $w = \varepsilon$]]
2. Para $i = 1$ até n , [[examina cada subcadeia de comprimento 1]]
3. Para cada variável A ,
4. Teste se $A \rightarrow b$, é uma regra, onde $b = w_i$.
5. Se for, coloque A na *tabela*(i, i).
6. Para $l = 2$ até n , [[l é o comprimento da subcadeia]]
7. Para $i = 1$ até $n - l + 1$, [[i é a posição inicial da subcadeia]]
8. Faça $j = i + l - 1$, [[j é a posição final da subcadeia]]
9. Para $k = i$ até $j - 1$, [[k é a posição onde a divisão ocorre]]
10. Para cada regra $A \rightarrow BC$,
11. Se *tabela*(i, k) contém B e *tabela*($k + 1, j$) contém C ,
ponha A na *tabela*(i, j).
12. Se S está em *tabela*($1, n$), *aceite*. Caso contrário, *rejeite*.”

Agora analisamos D . Cada estágio é facilmente implementado para rodar em tempo polinomial. Os estágios 4 e 5 rodam em no máximo nv vezes, onde v é o número de variáveis em G e é uma constante fixa independente de n ; daí esses estágios rodam $O(n)$ vezes. O estágio 6 rodam no máximo n vezes. Cada vez que o estágio 6 roda, o estágio 7 roda no máximo n vezes. Cada vez que o estágio 7 roda, os estágios 8 e 9 rodam no máximo n vezes. Cada vez que o estágio 9 roda, o estágio 10 roda r vezes, onde r é o número de regras de G e é uma outra constante fixa. Por conseguinte o estágio 11, o laço mais interno do algoritmo, roda $O(n^3)$ vezes. Somando o total mostra que D executa $O(n^3)$ estágios.

.....

7.3 A classe NP

Como observamos na Seção 7.2, podemos evitar a busca por força-bruta em muitos problemas e obter soluções de tempo polinomial. Entretanto, tentativas de evitar a força bruta em alguns outros problemas, incluindo muitos problemas interessantes e úteis, não têm sido bem sucedidas, e algoritmos de tempo polinomial que os resolvem não se sabe se existem.

Por que temos tido insucesso em encontrar algoritmos polinomiais para esses problemas? Não sabemos a resposta para essa importante questão. Talvez esses problemas têm algoritmos polinomiais, até agora não descobertos, que repousam sobre princípios desconhecidos. Ou possivelmente alguns desses problemas simplesmente *não* podem ser resolvidos em tempo polinomial. Eles podem ser intrinsecamente difíceis.

Uma descoberta impressionante concernente a essa questão mostra que as complexidades de muitos problemas estão ligadas. A descoberta de um algoritmo de tempo polinomial para um tal problema pode ser usada para resolver uma classe inteira de problemas. Para entender esse fenômeno, vamos começar com um exemplo.

Um **caminho hamiltoniano** em um grafo direcionado G é um caminho direcionado que passa por cada nó exatamente uma vez. Consideramos o problema de se testar se um grafo direcionado contém um caminho hamiltoniano conectando dois nós específicos, como mostrado na Figura 7.3. Seja

$$\begin{aligned} \text{CAMHAMIL} = & \{ \langle G, s, t \rangle \mid G \text{ é um grafo direcionado} \\ & \text{com um caminho hamiltoniano de } s \text{ para } t \}. \end{aligned}$$

Figura 7.3: Um caminho hamiltoniano passa por todo nó exatamente uma vez

Podemos facilmente obter um algoritmo de tempo exponencial para o problema *CAMHAMIL* modificando o algoritmo por força-bruta para *CAMINHO* dado no Teorema 7.12. Precisamos apenas adicionar um teste para verificar se o caminho em potencial é hamiltoniano. Ninguém sabe se *CAMHAMIL* é solúvel em tempo polinomial.

O problema *CAMHAMIL* realmente tem uma característica chamada **verificabilidade polinomial** que é importante para entender sua complexidade. Muito embora não conheçamos uma maneira rápida (i.e., de tempo polinomial) de determinar se um grafo contém um caminho hamiltoniano, se tal caminho fosse descoberto de alguma forma (talvez usando o algoritmo de tempo exponencial), poderíamos facilmente convencer alguém de sua existência, simplesmente apresentando-o. Em outras palavras, *verificar* a existência de um caminho hamiltoniano pode ser muito mais fácil que *de-terminar* sua existência.

Um outro problema polinomialmente verificável é a propriedade de ser composto. Lembre-se que um número natural é **composto** se ele é o produto de dois inteiros maiores que 1 (i.e., um número composto é um número que não é primo). Seja

$$\text{COMPOSTOS} = \{x \mid x = pq, \text{ para inteiros } p, q > 1\}.$$

Embora não conheçamos nenhum algoritmo de tempo polinomial para decidir esse problema, podemos facilmente verificar que um número é composto—tudo que é necessário é um divisor daquele número.

Alguns problemas podem não ser verificáveis polinomialmente. Por exemplo, tome $\overline{CAMHAMIL}$, o complemento do problema $CAMHAMIL$. Mesmo se pudéssemos determinar (de alguma forma) que um grafo realmente *não* tem um caminho hamiltoniano, não conheceríamos uma maneira pela qual uma outra pessoa pudesse verificar sua não-existência usando o mesmo algoritmo de tempo exponencial inicialmente usado para fazer a própria determinação da não-existência. Uma definição formal segue.

Definição 7.15

Um *verificador* para uma linguagem A é um algoritmo V , onde

$$A = \{w \mid V \text{ aceita } \langle w, c \rangle \text{ para alguma cadeia } c\}.$$

Medimos o tempo de um verificador somente em termos do comprimento de w , portanto um *verificador de tempo polinomial* roda em tempo polinomial no comprimento de w . Uma linguagem A é ‘polinomialmente verificável’ se ela tem um verificador de tempo polinomial.

Um verificador usa informação adicional, representada pelo símbolo c na Definição 7.15, para verificar que uma cadeia w é um membro de A . Essa informação é chamada um *certificado*, ou *prova*, de pertinência a A . Observe que, para verificadores polinomiais, o certificado tem comprimento polinomial (no comprimento de w) porque isso é tudo que o verificador pode acessar no seu limitante de tempo. Vamos aplicar essa definição às linguagens $CAMHAMIL$ e $COMPOSTOS$.

Para o problema $CAMHAMIL$, um certificado para uma cadeia $\langle G, s, t \rangle \in CAMHAMIL$ simplesmente é o caminho hamiltoniano de s para t . Para o problema $COMPOSTOS$, um certificado para o número composto x simplesmente é um de seus divisores. Em ambos os casos o verificador pode checar em tempo polinomial que a entrada está na linguagem quando lhe é dado o certificado.

Definição 7.16

NP é a classe de linguagens que têm verificadores polinomiais.

A classe NP é importante porque ela contém muitos problemas de interesse prático. Da discussão precedente, ambos $CAMHAMIL$ e $COMPOSTOS$ são membros de NP. O termo NP vem do *tempo polinomial não-determinístico* e é derivado de uma caracterização alternativa por meio do uso de máquinas de Turing não-determinísticas de tempo polinomial.

O que segue é uma máquina de Turing não-determinística (MTN) que decide o problema $CAMHAMIL$ em tempo polinomial não-determinístico. Lembre-se que na Definição 7.9 definimos o tempo de uma máquina não-determinística como sendo o tempo usado pelo ramo mais longo de computação.

N = “Sobre a entrada $\langle G, s, t \rangle$, onde G é um grafo direcionado com nós s e t :

1. Escreva uma lista de m números, p_1, \dots, p_m onde m é o número de nós em G . Cada número na lista é não-deterministicamente selecionado como estando entre 1 e m .
2. Verifique se há repetições na lista. Se alguma for encontrada, *rejeite*.
3. Verifique se $s = p_1$ e $t = p_m$. Se um dos dois falha, *rejeite*.

4. Para cada i entre 1 e $m - 1$, verifique se (p_i, p_{i+1}) é uma aresta de G . Se quaisquer não forem, *rejeite*. Caso contrário, todos os testes foram bem sucedidos, portanto *aceite*.”

Para analisar esse algoritmo e verificar que ele roda em tempo polinomial não-determinístico, examinamos cada um dos seus estágios. No estágio 1, a escolha não-determinística claramente roda em tempo polinomial. Nos estágios 2 e 3, cada parte é uma simples verificação, portanto eles rodam em tempo polinomial. Finalmente, o estágio 4 também claramente roda em tempo polinomial. Por conseguinte esse algoritmo roda em tempo polinomial não-determinístico.

Teorema 7.17

Uma linguagem está em NP sse ela é decidida por alguma máquina de Turing não-determinística de tempo polinomial.

Idéia da prova. Mostramos como converter um verificador de tempo polinomial para uma MTN de tempo polinomial e vice-versa. A MT simula o verificador adivinhando o certificado. O verificador simula a MTN usando o ramo de aceitação como o certificado.

Prova. Para a direção de trás para frente deste teorema, suponha que $A \in \text{NP}$ e mostre que A é decidida por uma MTN de tempo polinomial N . Seja V o verificador de tempo polinomial para A que existe pela definição de NP. Assuma que V seja uma MT que roda em tempo n^k e construa N da seguinte forma:

N = “Sobre a entrada w de comprimento n ,

1. Não-deterministicamente selecione a cadeia c de comprimento n^k .
2. Rode V sobre a entrada $\langle w, c \rangle$.
3. Se V aceita, *aceite*; caso contrário, *rejeite*.”

Para provar a outra direção do teorema, assuma que A é decidida por uma MTN de tempo polinomial N e construa um verificador de tempo polinomial V da seguinte forma:

V = “Sobre a entrada $\langle w, c \rangle$, onde w e c são cadeias:

1. Simule N sobre a entrada w , tratando cada símbolo de c como uma descrição da escolha não-determinística a fazer a cada passo (como na prova do Teorema 3.10).
2. Se esse ramo da computação de N aceita, *aceite*; caso contrário, *rejeite*.”

Definimos a classe de complexidade de tempo não-determinístico $\text{NTIME}(t(n))$ como análoga à classe de complexidade de tempo determinístico $\text{TIME}(t(n))$.

Definição 7.18

$\text{NTIME}(t(n)) = \{L \mid L \text{ é uma linguagem decidida por uma máquina de Turing não-determinística de tempo } t(n)\}$.

Corolário 7.19
 $\text{NP} = \bigcup_k \text{NTIME}(n^k)$.

A classe NP é insensível à escolha de modelo computacional não-determinístico razoável porque todos tais modelos são polinomialmente equivalentes. Quando estamos descrevendo e analisando algoritmos de tempo polinomial não-determinístico seguimos as convenções precedentes para algoritmos de tempo polinomial determinístico. Cada estágio de um algoritmo de tempo polinomial não-determinístico tem que ter uma implementação óbvia em tempo polinomial não-determinístico em um modelo computacional não-determinístico razoável. Analisamos o algoritmo para mostrar que todo ramo usa no máximo uma quantidade polinomial de estágios.

Exemplos de problemas em NP

Um *clique* em um grafo não-direcionado é um subgrafo, no qual cada dois nós são conectados por uma aresta. Um *k-clique* é um clique que contém k nós. A Figura 7.4 ilustra um grafo tendo um 5-clique.

Figura 7.4: Um grafo com um 5-clique

O problema do clique é determinar se um grafo contém um clique de um tamanho especificado. Seja

$$\text{CLIQUE} = \{\langle G, k \rangle \mid G \text{ é um grafo não-direcionado com um } k\text{-clique}\}.$$

Teorema 7.20
 CLIQUE está em NP.

.....
Idéia da prova. O clique é o certificado.

Prova. O seguinte é um verificador V para CLIQUE .

V = “Sobre a entrada $\langle\langle G, k \rangle, c\rangle$:

1. Teste se c é um conjunto de k nós em G .
2. Teste se G contém todas as arestas conectando nós em c .
3. Se ambos passam, *aceite*; caso contrário, *rejeite*.”

Prova alternativa. Se você prefere pensar em NP em termos de máquinas de Turing de tempo polinomial não-determinístico, você pode provar esse teorema fornecendo um que decida CLIQUE . Observe a similaridade entre as duas provas.

N = “Sobre a entrada $\langle G, k \rangle$, onde G é um grafo:

1. Não-deterministicamente selecione um subconjunto c de k nós de g .
2. Teste se G contém todas as arestas conectando os nós em c .

3. Se sim, *aceite*; caso contrário, *rejeite*.”
-

A seguir consideramos o problema *SOMA-SUBCONJ* concernente à aritmética inteira. Nesse problema temos uma coleção de números, x_1, \dots, x_k e um número alvo t . Desejamos determinar se a coleção contém uma subcoleção cuja soma é t . Por conseguinte

$$\text{SOMA-SUBCONJ} = \{\langle S, t \rangle \mid S = \{x_1, \dots, x_k\} \text{ e para algum } \{y_1, \dots, y_l\} \subseteq \{x_1, \dots, x_k\}, \text{ temos } \sum y_i = t\}.$$

Por exemplo, $\langle \{4, 11, 16, 21, 27\}, 25 \rangle \in \text{SOMA-SUBCONJ}$ porque $4 + 21 = 25$. Note que $\{x_1, \dots, x_k\}$ e $\{y_1, \dots, y_l\}$ são considerados como sendo ***multiconjuntos*** e portanto permitem repetições de elementos.

Teorema 7.21
SOMA-SUBCONJ está em NP.

Idéia da prova. O subconjunto é o certificado.

Prova. O seguinte é um verificador V para *SOMA-SUBCONJ*.

V = “Sobre a entrada $\langle \langle S, t \rangle, c \rangle$:

1. Teste se c é uma coleção de números cuja soma é t .
2. Teste se S contém todos os números em c .
3. Se ambos passam, *aceite*; caso contrário, *rejeite*.”

Prova alternativa. Podemos também provar esse teorema dando uma máquina de Turing de tempo polinomial não-determinístico para *SOMA-SUBCONJ* da seguinte forma.

N = “Sobre a entrada $\langle S, t \rangle$:

1. Não-deterministicamente selecione um subconjunto c dos números em S .
 2. Teste se c é uma coleção de números cuja soma é t .
 3. Se o teste passa, *aceite*; caso contrário, *rejeite*.”
-

Observe que os complementos desses conjuntos, *CLIQUE* e *SOMA-SUBCONJ*, não são obviamente membros de NP. Verificar que algo *não* está presente parece ser mais difícil que verificar que *está* presente. Montamos uma classe de complexidade separada, chamada coNP, que contém as linguagens que são complementos de linguagens em NP. Não sabemos se coNP é diferente de NP.

A questão P versus NP

Como temos estado dizendo, NP é a classe de linguagens que são solúveis em tempo polinomial numa máquina de Turing não-determinística, ou, equivalentemente, é a classe de linguagens na qual pertinência na linguagem pode ser verificada em tempo polinomial. P é a classe de linguagens nas quais pertinência pode ser testada em tempo polinomial. Resumimos essa informação da seguinte forma, onde nos referimos frouxamente a solúvel em tempo polinomial como solúvel “rapidamente.”

P = a classe de linguagens nas quais pertinência pode ser *decidida* rapidamente.

NP = a classe de linguagens nas quais pertinência pode ser *verificada* rapidamente.

Apresentamos exemplos de linguagens, tais como *CAMHAMIL* e *CLIQUE*, que são membros de NP mas que não se sabe se estão em P. O poder de verificabilidade polinomial parece ser muito maior que aquele de decidibilidade polinomial. Mas, por mais difícil que seja de imaginar, P e NP poderiam ser iguais. Estamos incapacitados de *provar* a existência de uma única linguagem em NP que não esteja em P.

A questão de se P=NP é um dos maiores problemas ainda não resolvidos em ciência da computação teórica e matemática contemporânea. Se essas classes fossem iguais, qualquer problema polinomialmente verificável seria polinomialmente decidível. A maioria dos pesquisadores acreditam que as duas classes não são iguais porque as pessoas têm investido enormes esforços para encontrar algoritmos de tempo polinomial para certos problemas em NP, sem sucesso. Pesquisadores também têm tentado provar que as classes são desiguais, mas isso acarretaria mostrar que nenhum algoritmo rápido existe para substituir a busca por força-bruta. Fazer isso está atualmente além do alcance científico. A Figura 7.5 mostra as duas possibilidades.

Figura 7.5: Uma dessas possibilidades é correta

O melhor método conhecido para resolver linguagens em NP deterministicamente usa tempo exponencial. Em outras palavras, podemos provar que

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}),$$

mas não sabemos se NP está contida em uma classe de complexidade de tempo determinístico menor.

7.4 NP-completude

Um importante avanço na questão P versus NP veio no início dos anos 1970's com o trabalho de Stephen Cook e Leonid Levin. Eles descobriram certos problemas em NP cuja complexidade individual está relacionada àquela da classe inteira. Se um algoritmo de tempo polinomial existe para algum desses problemas, todos os problemas em NP seriam solúveis em tempo polinomial. Esses problemas são chamados **NP-completos**. O fenômeno de NP-completude é importante tanto por razões práticas quanto teóricas.

No lado teórico, um pesquisador tentando mostrar que P é diferente de NP pode se concentrar num problema NP-completo. Se qualquer problema em NP requer mais

que tempo polinomial, um NP-completo também requer. Ademais, um pesquisador tentando provar que P é igual a NP precisa somente encontrar um algoritmo de tempo polinomial para um problema NP-completo para atingir seu objetivo.

No lado prático, o fenômeno da NP-completude pode evitar o desperdício de tempo na busca por um algoritmo de tempo polinomial não existente para resolver um problema específico. Muito embora possivelmente não tenhamos a matemática necessária para provar que o problema é insolúvel em tempo polinomial, acreditamos que P é diferente de NP , portanto provar que um problema é NP-completo é forte evidência de sua não-polinomialidade.

O primeiro problema NP-completo que apresentamos é chamado o **problema da satisfatibilidade**. Lembre-se que variáveis que podem tomar valores VERDADEIRO e FALSO são chamadas **variáveis booleanas** (veja a Seção 0.2). Usualmente, representamos VERDADEIRO por 1 e FALSO por 0. As **operações booleanas** E, OU, e NÃO, representadas pelo símbolos \wedge , \vee , e \neg , respectivamente, são descritas na lista abaixo. Usamos a barra superior como uma abreviação para o símbolo \neg , portanto \bar{x} significa $\neg x$.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \bar{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \bar{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

Uma **fórmula booleana** é uma expressão envolvendo variáveis booleanas e operações. Por exemplo,

$$\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$$

é uma fórmula booleana. Uma fórmula booleana é **satisfatível** se alguma atribuição de 0's e 1's às variáveis faz com que o valor da fórmula seja 1. A fórmula precedente é satisfatível porque a atribuição $x = 0$, $y = 1$, e $z = 0$ faz com que ϕ tenha valor 1. Dizemos que a atribuição *satisfaz* ϕ . O **problema da satisfatibilidade** é testar se uma fórmula booleana é satisfatível. Seja

$$SAT = \{\langle \phi \rangle \mid \phi \text{ é uma fórmula booleana}\}.$$

Agora enunciamos o teorema de Cook–Levin que liga a complexidade do problema SAT às complexidades de todos os problemas em NP .

Teorema 7.22

Teorema de Cook–Levin $SAT \in P$ sse $P=NP$.

A seguir, desenvolvemos o método que é central para a prova do teorema de Cook–Levin.

Redutibilidade em tempo polinomial

No Capítulo 5 definimos o conceito de reduzir um problema a outro. Quando o problema A se reduz ao problema B , uma solução para B pode ser usada para resolver A . Agora definimos uma versão de redutibilidade que leva em conta a eficiência da computação. Quando o problema A é *eficientemente* redutível ao problema B , uma solução eficiente para B pode ser usada para resolver A eficientemente.

Definição 7.23

Uma função $f : \Sigma^* \rightarrow \Sigma^*$ é uma **função computável em tempo polinomial** se alguma máquina de Turing de tempo polinomial M existe que pára com exatamente $f(w)$ sobre sua fita, quando iniciada com qualquer entrada w .

Definição 7.24

A linguagem A é **redutível por mapeamento em tempo polinomial**,¹ em alguns outros livros-texto, ou simplesmente **redutível em tempo polinomial**, à linguagem B , escrito $A \leq_P B$, se uma função computável em tempo polinomial $f : \Sigma^* \rightarrow \Sigma^*$ existe, onde para toda w , $w \in A \iff f(w) \in B$. A função f é chamada a **redução de tempo polinomial** de A para B .

Redutibilidade em tempo polinomial é o análogo eficiente à redutibilidade por mapeamento, como definido na Seção 5.3. Outras formas de redutibilidade eficiente estão disponíveis, mas redutibilidade polinomial é uma forma simples que é adequada para nossos propósitos portanto não discutiremos as outras aqui. A Figura 7.6 ilustra a redutibilidade em tempo polinomial.

Figura 7.6: Função de tempo polinomial f reduzindo A para B

Como na redução por mapeamento comum, uma redução em tempo polinomial de A para B provê uma maneira de converter teste de pertinência em A para teste de pertinência em B , mas agora a conversão é feita eficientemente. Para testar se $w \in A$, usamos a redução f para mapear w para $f(w)$ e testamos se $f(w) \in B$.

Se uma linguagem é redutível em tempo polinomial para uma linguagem para a qual já se sabe que tem uma solução polinomial, obtemos uma solução polinomial para a linguagem original, como no seguinte teorema.

Teorema 7.25

Se $A \leq_P B$ e $B \in P$, então $A \in P$.

Prova. Seja M o algoritmo de tempo polinomial que decide B e f a redução em tempo polinomial de A para B . Descrevemos um algoritmo de tempo polinomial N que decide A da seguinte forma.

N = “Sobre a entrada w :

1. Compute $f(w)$.
2. Rode M sobre a entrada $f(w)$ e dê como saída o que quer que M dê como saída.”

Se $w \in A$, então $f(w) \in B$ porque f é uma redução de A para B . Por conseguinte M aceita $f(w)$ sempre que $w \in A$. Além do mais, N roda em tempo polinomial porque cada um dos seus dois estágios roda em tempo polinomial. Note que o estágio 2 roda em tempo polinomial porque a composição de dois polinômios é um polinômio.

¹É chamada redutibilidade muitos-para-um em tempo polinomial

Antes de demonstrar uma redução em tempo polinomial introduzimos $3SAT$, um caso especial do problema da satisfatibilidade no qual todas as fórmulas estão numa forma especial. Um **literal** é uma variável booleana ou uma variável booleana negada, como em x ou \bar{x} . Uma **cláusula** é vários literais conectados com \vee 's, como em $(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4)$. Uma fórmula booleana está na **forma normal conjuntiva**, chamada **fnc-fórmula**, se ela compreende várias cláusulas conectadas com \wedge 's como em

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_3 \vee \bar{x}_6).$$

Ela é uma **3fnc-fórmula** se todas as cláusulas têm três literais, como em

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_5 \vee x_6) \wedge (x_3 \vee \bar{x}_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Seja $3SAT = \{\langle \phi \rangle \mid \phi \text{ é uma 3fnc-fórmula satisfatível}\}$. Em uma fnc-fórmula, cada cláusula tem que conter pelo menos um literal que recebe o valor 1.

O teorema seguinte apresenta uma redução em tempo polinomial do problema $3SAT$ para o problema $CLIQUE$.

Teorema 7.26
 $3SAT$ é redutível em tempo polinomial para $CLIQUE$.

Idéia da prova. A redução em tempo polinomial f que demonstramos de $3SAT$ para $CLIQUE$ converte fórmulas em grafos. Nos grafos construídos, cliques de um tamanho especificado correspondem a atribuições que satisfazem à fórmula. Estruturas dentro do grafo são projetadas para reproduzir o comportamento das variáveis e das cláusulas.

Prova. Seja ϕ uma fórmula com k cláusulas tal como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \dots \wedge (a_k \vee b_k \vee c_k).$$

A redução f gera a cadeia $\langle G, k \rangle$, onde G é um grafo não-direcionado definido como segue.

Os nós em G conectam todos exceto dois tipos de pares de nós em G . Nenhuma aresta está presente entre nós na mesma tripla e nenhuma aresta está presente entre dois nós com rótulos contraditórios, como em x_2 e \bar{x}_2 . A Figura 7.7 ilustra essa construção quando $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$.

Figura 7.7: O grafo que a redução produz a partir de $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$

Agora demonstramos por que essa construção funciona. Mostramos que ϕ é satisfatível sse G tem um k -clique.

Suponha que ϕ tenha uma atribuição que a satisfaz. Nessa atribuição, pelo menos um literal é verdadeiro em toda cláusula. Em cada tripla de G , selecionamos um nó correspondendo a um literal verdadeiro na atribuição que satisfaz a fórmula. Se mais

de um literal é verdadeiro em uma cláusula específica, escolhemos um dos literais verdadeiros arbitrariamente. Os nós selecionados formam um k -clique. O número de nós selecionados é k , porque escolhemos um para cada uma das k triplas. Cada par de nós selecionados é ligado por uma aresta porque nenhum par se encaixa numa das exceções descritas anteriormente. Eles não poderiam ser da mesma tripla porque selecionamos apenas um nó por tripla. Eles não poderiam ter rótulos contraditórios porque os literais associados eram ambos verdadeiros na atribuição que satisfaz a fórmula. Por conseguinte G é um k -clique.

Suponha que G tenha um k -clique. Não é o caso que dois dos nós do clique ocorrem na mesma tripla porque nós na mesma tripla não estão conectados por arestas. Por conseguinte cada uma das k triplas contém exatamente um dos nós do k clique. Atribuimos valores-verdade às variáveis de ϕ de modo que cada literal rotulando um nó do clique é tornado verdadeiro. Fazer isso é sempre possível porque dois nós rotulados de uma maneira contraditória não são conectados por uma aresta e portanto ambos não podem estar no clique. Essa atribuição às variáveis satisfaz ϕ porque cada tripla contém um nó do clique e portanto cada cláusula contém um literal que é atribuído VERDADEIRO. Por conseguinte ϕ é satisfatível.

.....

Os teoremas 7.25 e 5.26 nos dizem que, se *CLIQUE* for solúvel em tempo polinomial, então *3SAT* também o é. À primeira vista, essa conexão entre esses dois problemas parece um tanto notável porque, superficialmente, eles são bastante diferentes. Mas redutibilidade em tempo polinomial nos permite ligar suas complexidades. Agora nos voltamos para uma definição que nos permitirá ligar as complexidades de uma classe inteira de problemas.

Definição de NP-completude

Definição 7.27

Uma linguagem B é **NP-completa** se ela satisfaz duas condições:

1. B está em NP, e
2. toda A em NP é redutível em tempo polinomial a B .

Teorema 7.28

Se B é NP-completa e $B \in \text{NP}$, então $P=NP$.

.....

Prova. Esse teorema segue diretamente da definição de redutibilidade em tempo polinomial.

.....

Teorema 7.29

Se B é NP-completa e $B \leq_P C$ para C em NP, então C é NP-completa.

Prova. Já sabemos que C está em NP, portanto temos que mostrar que toda A que está em NP é redutível em tempo polinomial a C . Devido ao fato de que B é NP-completa, toda linguagem em NP é redutível em tempo polinomial a B , e B por sua vez é redutível em tempo polinomial a C . Reduções em tempo polinomial compõem; ou seja, se A é redutível em tempo polinomial a C e C é redutível em tempo polinomial

a B , então A é redutível em tempo polinomial a B . Daí toda linguagem em NP é redutível em tempo polinomial a C .

O teorema de Cook–Levin

Uma vez que temos um problema NP-completo, podemos obter outros por redução em tempo polinomial a partir do primeiro. Entretanto, estabelecer o primeiro problema NP-completo é mais difícil. Agora fazemos isso provando que SAT é NP-completo.

Teorema 7.30
 SAT é NP-completo.²

Esse teorema reenuncia o Teorema 7.22, o teorema de Cook–Levin, de uma outra forma.

Idéia da prova. Mostrar que SAT está em NP é fácil, e fazemos isso logo mais. A parte difícil da prova é mostrar que qualquer linguagem em NP é redutível em tempo polinomial a SAT .

Para fazer isso construímos uma redução em tempo polinomial para cada linguagem A em NP para SAT . A redução para A toma uma cadeia w e produz uma fórmula booleana ϕ que simula a máquina NP para A sobre a entrada w . Se a máquina aceita, ϕ tem uma atribuição que satisfaz correspondendo à computação de aceitação. Se a máquina não aceita, nenhuma atribuição satisfaz ϕ . Por conseguinte, w está em A se e somente se ϕ é satisfatível.

Construir verdadeiramente a redução para funcionar dessa maneira é uma tarefa conceitualmente simples, embora tenhamos que lidar com muitos detalhes. Uma fórmula booleana pode conter as operações booleanas E, OU, e NÃO, e essas operações formam a base para a circuitaria usada em computadores eletrônicos. Portanto, o fato de que podemos desenhar uma fórmula booleana para simular uma máquina de Turing não é surpreendente. Os detalhes estão na implementação dessa idéia.

Prova. Primeiro, mostramos que SAT está em NP. Uma máquina de tempo polinomial não-determinístico pode adivinhar uma atribuição para uma dada fórmula ϕ e aceitar se a atribuição satisfaz ϕ .

A seguir, tomamos qualquer linguagem A em NP e mostramos que A é redutível em tempo polinomial a SAT . Seja N uma máquina de Turing não-determinística que decide A em tempo n^k para alguma constante k . (Por conveniência assumimos na verdade que N roda em tempo $n^k - 3$, mas somente aqueles leitores interessados em detalhes devem se preocupar com esse pequeno detalhe.) A seguinte noção ajuda a descrever a redução.

Um **tableau** para N sobre w é uma tabela $n^k \times n^k$ cujas linhas são as configurações de um ramo da computação de N sobre a entrada w , como mostrado na Figura 7.8. Por conveniência mais adiante assumimos que cada configuração começa e termina com um símbolo $\#$, portanto a primeira e a última coluna de um tableau são todas de $\#$'s. A primeira linha do tableau é a configuração inicial de N sobre w , e cada linha segue da anterior conforme a função de transição de N . Um tabelau é de **aceitação** se qualquer linha do tableau é uma configuração de aceitação.

Figura 7.8: Um tableau é uma tabela $n^k \times n^k$ de configurações

Todo tableau de aceitação para N sobre w corresponde a um ramo da computação de N sobre w . Por conseguinte o problema de se determinar se N aceita w é equivalente ao problema de se determinar se um tableau de aceitação para N sobre w existe.

Agora chegamos à descrição da redução em tempo polinomial f de A para SAT . Sobre a entrada w , a redução produz uma fórmula ϕ . Começamos descrevendo as variáveis de ϕ . Digamos que Q e Γ sejam o conjunto de estados e o alfabeto de fita de N . Seja $C = Q \cup \Gamma \cup \{\#\}$. Para cada i e j entre 1 e n^k e para cada s em C temos uma variável $x_{i,j,s}$.

Cada uma das $(n^k)^2$ entradas de um tableau é chamado de uma **célula**. A célula na linha i e coluna j é chamada $celula[i, j]$ e contém um símbolo de C . Representamos o conteúdo das células com as variáveis de ϕ . Se $x_{i,j,s}$ toma o valor 1, isso significa que $celula[i, j]$ contém um s .

Agora desenhamos ϕ de modo que uma atribuição às variáveis que satisfaça a fórmula corresponde a um tableau de aceitação para N sobre w . A fórmula ϕ é o E das quatro partes $\phi_{celula} \wedge \phi_{inicio} \wedge \phi_{movimento} \wedge \phi_{aceita}$ e descrevemos cada um por vez.

Como mencionamos anteriormente, ligar uma variável $x_{i,j,s}$ corresponde a colocar um símbolo s em $celula[i, j]$. A primeira coisa que temos que garantir de modo a obter uma correspondência entre uma atribuição e um tableau é que a atribuição liga exatamente uma variável para cada célula. A fórmula ϕ_{celula} assegura esse requisito expressando-o em termos de operações booleanas:

$$\phi_{celula} = \bigwedge_{1 \leq i, j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigvee_{\substack{s, t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

Os símbolos \wedge e \vee significam iteração de E's e OU's respectivamente. Por exemplo, o fragmento da fórmula anterior

$$\bigvee_{s \in C} x_{i,j,s}$$

é uma abreviação de

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \cdots \vee x_{i,j,s_l}$$

onde $C = \{s_1, s_2, \dots, s_l\}$. Portanto, ϕ_{celula} é na verdade uma expressão grande que contém um fragmento para cada célula no tableau porque i e j variam de 1 a n^k . A primeira parte de cada fragmento diz que pelo menos uma variável é ligada na célula correspondente. A segunda parte de cada fragmento diz que não mais que uma variável é ligada (literalmente, ela diz que em cada par de variáveis, pelo menos uma é ligada) na célula correspondente. Esses fragmentos são conectados por operações \wedge .

A primeira parte de ϕ_{celula} dentro dos colchetes estipula que pelo menos uma variável que está associada a cada célula está ligada, enquanto que a segunda parte estipula que não mais que uma variável está ligada para cada célula. Qualquer atribuição às

²Uma prova alternativa desse teorema aparece na Seção 9.3 na página 321.

variáveis que satisfaz ϕ e por conseguinte ϕ_{celula} tem que ter exatamente uma variável ligada para toda célula. Por conseguinte qualquer atribuição que satisfaz a fórmula especifica um símbolo em cada célula da tabela. As partes $\phi_{início}$, $\phi_{movimento}$, e ϕ_{aceita} asseguram que a tabela é na verdade um tableau de aceitação da seguinte forma.

A fórmula $\phi_{início}$ assegura que a primeira linha da tabela é a configuração inicial de N sobre w explicitamente estipulando que as variáveis correspondentes estão ligadas:

$$\begin{aligned}\phi_{início} = & \quad x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \dots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \dots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}.\end{aligned}$$

A fórmula ϕ_{aceita} garante que uma configuração de aceitação ocorre no tableau. Ela assegura que q_{aceita} , o símbolo para o estado de aceitação, aparece em uma das células do tableau, estipulando que uma das variáveis correspondentes está ligada:

$$\phi_{aceita} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{aceita}}.$$

Finalmente, a fórmula $q_{movimento}$ garante que cada linha da tabela corresponde a uma configuração que legalmente segue da configuração da linha precedente conforme as regras de N . Ela faz isso assegurando que cada janela 2×3 de células é legal. Dizemos que uma janela 2×3 é *legal* se aquela janela não viola as ações especificadas pela função de transição de N . Em outras palavras, uma janela é legal se ela pode aparecer quando uma configuração corretamente segue uma outra.³

Por exemplo, digamos que a , b , e c são membros do alfabeto de fita e q_1 e q_2 são estados de N . Assuma que quando no estado q_1 com a cabeça lendo um a , N escreve um b , permanece no estado q_1 e move para a direita, e que quando no estado q_1 com a cabeça lendo um b , N não-deterministicamente

1. escreve um c , vai para q_2 e move para a esquerda, ou
2. escreve um a , vai para q_2 e move para a direita.

Expressando formalmente, $\delta(q_1, a) = \{(q_1, b, D)\}$ e $\delta(q_1, b) = \{(q_2, c, E), (q_2, a, D)\}$. Exemplos de janelas legais para essa máquina são mostrados na Figura 7.9.

(a)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>q_1</td><td>b</td></tr><tr><td>q_2</td><td>a</td><td>c</td></tr></table>	a	q_1	b	q_2	a	c	(b)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>q_1</td><td>b</td></tr><tr><td>a</td><td>a</td><td>q_2</td></tr></table>	a	q_1	b	a	a	q_2	(c)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>a</td><td>q_1</td></tr><tr><td>a</td><td>a</td><td>b</td></tr></table>	a	a	q_1	a	a	b
a	q_1	b																					
q_2	a	c																					
a	q_1	b																					
a	a	q_2																					
a	a	q_1																					
a	a	b																					
(d)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>#</td><td>b</td><td>a</td></tr><tr><td>#</td><td>b</td><td>a</td></tr></table>	#	b	a	#	b	a	(e)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>a</td><td>b</td><td>a</td></tr><tr><td>a</td><td>b</td><td>q_2</td></tr></table>	a	b	a	a	b	q_2	(f)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>b</td><td>b</td><td>b</td></tr><tr><td>c</td><td>b</td><td>b</td></tr></table>	b	b	b	c	b	b
#	b	a																					
#	b	a																					
a	b	a																					
a	b	q_2																					
b	b	b																					
c	b	b																					

Figura 7.9: Exemplos de janelas legais

³Poderíamos dar uma definição precisa de *janela legal* aqui, em termos da função de transição. Mas fazer isso é bastante entendente e seria uma distração da principal linha da argumentação. Qualquer um que deseje mais precisão deve se remeter à análise relacionada na prova do Teorema 5.11, a indecidibilidade do Problema da Correspondência de Post.

Na Figura 7.9, as janelas (a) e (b) são legais porque a função de transição permite que N se move da maneira indicada. A janela (c) é legal porque, com q_1 aparecendo no lado direito da linha superior, não sabemos sobre que símbolo a cabeça está. Esse símbolo poderia ser um a , e q_1 poderia modificá-lo para um b e mover para a direita. Essa possibilidade daria origem a essa janela, portanto ela não viola as regras de N . A janela (d) é obviamente legal porque as linhas superior e inferior são idênticas, o que ocorreria se a cabeça não estivesse adjacente à localização da janela. Note que um $\#$ pode aparecer na esquerda ou na direita de ambas as linhas superior e inferior em uma janela legal. A janela (e) é legal porque o estado q_1 lendo um b poderia ter estado imediatamente à direita da linha superior, e ela teria então movido para a esquerda no estado q_2 para aparecer na extremidade direita da linha inferior. Finalmente, a janela (f) é legal porque o estado q_1 poderia ter estado imediatamente à esquerda da linha superior e ela poderia ter modificado o b para um c e movido para a esquerda.

As janelas mostradas na Figura 7.10 não são legais para a máquina N .

(a)	<table border="1"><tr><td>a</td><td>b</td><td>a</td></tr><tr><td>a</td><td>a</td><td>a</td></tr></table>	a	b	a	a	a	a	(b)	<table border="1"><tr><td>a</td><td>q_1</td><td>b</td></tr><tr><td>q_1</td><td>a</td><td>a</td></tr></table>	a	q_1	b	q_1	a	a	(c)	<table border="1"><tr><td>b</td><td>q_1</td><td>b</td></tr><tr><td>q_2</td><td>b</td><td>q_2</td></tr></table>	b	q_1	b	q_2	b	q_2
a	b	a																					
a	a	a																					
a	q_1	b																					
q_1	a	a																					
b	q_1	b																					
q_2	b	q_2																					

Figura 7.10: Exemplos de janelas ilegais

Na janela (a) o símbolo central na linha superior não pode modificar porque um estado não estava adjacente a ele. A janela (b) não é legal porque a função de transição especifica que o b é modificado para um c mas não para um a . A janela (c) não é legal porque dois estados aparecem na linha inferior.

Afirmção 7.31

Se a linha superior da tabela é a configuração inicial e toda janela na tabela é legal, cada linha da tabela é uma configuração que segue legalmente a anterior.

Provamos essa afirmação considerando quaisquer duas configurações adjacentes na tabela, chamadas configuração superior e configuração inferior. Na configuração superior, toda célula que não é adjacente a um símbolo de estado e que não contém o símbolo de fronteira $\#$, é a célula central superior em uma janela cuja linha superior não contém estados. Por conseguinte aquele símbolo tem que aparecer o mesmo na célula central inferior da janela. Por conseguinte ele aparece na mesma posição na configuração inferior.

A janela contendo o símbolo de estado na célula central superior garante que as três posições correspondentes são atualizadas consistentemente com a função de transição. Por conseguinte, se a configuração superior é uma configuração legal, assim o é a configuração inferior, e a inferior segue a superior conforme as regras de N . Note que esta prova, embora simples, depende crucialmente da nossa escolha de um tamanho de janela de 2×3 , como mostra o Exercício 7.32.

Agora voltamos à construção de $\phi_{\text{movimento}}$. Ela estipula que todas as janelas no tableau são legais. Cada janela contém seis células, que podem ser arranjadas de um número fixo de maneiras para produzir uma janela legal. A fórmula $\phi_{\text{movimento}}$ diz que os arranjos daquelas células tem que ser uma dessas maneiras, ou

$$\phi_{\text{movimento}} = \bigwedge_{1 < i \leq n^k, 1 < j < n^k} (\text{a janela } (i, j) \text{ é legal})$$

Substituimos o texto “a janela (i, j) é legal” nessa fórmula pela fórmula a seguir. Escrevemos o conteúdo das seis células de uma janela como a_1, \dots, a_6 .

$$\bigvee_{\substack{a_1, \dots, a_6 \\ \text{é uma janela legal}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

A seguir analisamos a complexidade da redução para mostrar que ela opera em tempo polinomial. Para fazer isso examinamos o tamanho de ϕ . Lembre-se que o tableau é uma tabela de $n^k \times n^k$, portanto ele contém n^{2k} células. Cada célula tem l variáveis associadas a ela, o número total de variáveis é $O(n^{2k})$.

A fórmula ϕ_{celula} contém um fragmento de tamanho-fixo da fórmula para cada célula do tableau, portanto seu tamanho é $O(n^{2k})$. A fórmula ϕ_{inicio} tem um fragmento para cada célula na linha superior, portanto seu tamanho é $O(n^k)$. As fórmulas $\phi_{\text{movimento}}$ e ϕ_{aceita} cada uma contém um fragmento de tamanho-fixo da fórmula para cada célula do tableau, portanto seu tamanho é $O(n^{2k})$. Por conseguinte o tamanho total de ϕ é $O(n^{2k})$. Esse resultado é bom porque o tamanho de ϕ é polinomial em n . Se ele fosse mais que polinomial, a redução não teria qualquer chance de gerá-la em tempo polinomial. (Na realidade nossas estimativas são baixas por um fator de $O(\log n)$ porque cada variável tem índices que podem variar até n^k e portanto podem requerer $O(\log n)$ símbolos para escrever nas fórmulas, mas esse fator adicional não modifica a polinomialidade do resultado.

Para ver que podemos gerar a fórmula em tempo polinomial, observe sua natureza altamente repetitiva. Cada componente da fórmula é composto de muitos fragmentos quase idênticos, que diferem apenas nos índices de uma maneira simples. Por conseguinte podemos facilmente construir uma redução que produz ϕ em tempo polinomial a partir da entrada w .

Portanto concluímos a prova do teorema de Cook–Levin, mostrando que a linguagem SAT é NP-completa. Mostrar a NP-completude de outras linguagens geralmente não requer uma prova tão longa. Ao invés disso, NP-completude pode ser provada com uma redução em tempo polinomial a partir de uma linguagem que já é conhecida ser NP-completa. Podemos usar SAT para esse propósito, mas usar $3SAT$, o caso especial de SAT que definimos na página 251, é usualmente mais fácil. Lembre-se que as fórmulas em $3SAT$ estão na forma normal conjuntiva (fnc) com três literais por cláusula. Primeiro, temos que mostrar que $3SAT$ propriamente dita é NP-completa. Provamos isso como um corolário do Teorema 7.30.

Corolário 7.32
 $3SAT$ é NP-completa.

Prova. Obviamente $3SAT$ está em NP, portanto só precisamos provar que todas as linguagens em NP se reduzem a $3SAT$ em tempo polinomial. Uma maneira de fazer isso é mostrar que SAT se reduz em tempo polinomial a $3SAT$. Ao invés disso, modificamos a prova do Teorema 7.30 de modo que ela produza diretamente uma fórmula na forma normal conjuntiva com três literais por cláusula.

O Teorema 7.30 produz uma fórmula que já está quase na forma normal conjuntiva. A fórmula ϕ_{celula} é um grande E de subfórmulas, cada uma das quais contém um grande OU e um grande E de OU's. Por conseguinte ϕ_{celula} é um E de cláusulas e portanto já está na fnc. A fórmula ϕ_{inicio} é um grande E de variáveis. Tomando cada uma dessas variáveis como sendo uma cláusula de tamanho 1 vemos que ϕ_{inicio} está na fnc. A fórmula ϕ_{aceita} é um grande OU de variáveis e portanto é uma única cláusula. A fórmula $\phi_{movimento}$ é a única que não já está na fnc, mas podemos facilmente convertê-la em uma fórmula que está na fnc da seguinte maneira.

Lembre-se que $\phi_{movimento}$ é um grande E de subfórmulas, cada uma das quais é um OU de E's que descreve todas as possíveis janelas legais. As leis de distributividade, conforme descritas no Capítulo 0, enunciam que podemos substituir um OU de E's por um E de OU's equivalente. Fazendo isso aumenta significativamente o tamanho de cada subfórmula, mas só pode aumentar o tamanho total de $\phi_{movimento}$ por um fator constante porque o tamanho de cada subfórmula depende somente de N . O resultado é uma fórmula que está na forma normal conjuntiva.

Agora que escrevemos a fórmula na fnc, convertemo-la para uma com três literais por cláusula. Em cada cláusula que no momento tem um ou dois literais, replicamos um dos literais até que o número total seja três. Em cada cláusula que tem mais de três literais, dividimo-la em várias cláusulas e acrescentamos variáveis adicionais para preservar a satisfatibilidade ou não-satisfatibilidade da original.

Por exemplo, substituimos a cláusula $(a_1 \vee a_2 \vee a_3 \vee a_4)$ na qual cada a_i é um literal pela expressão composta de duas cláusulas $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$ na qual z é uma nova variável. Se alguma valoração dos a_i 's satisfaz a cláusula original, podemos encontrar alguma valoração de z de modo que as duas novas cláusulas são satisfeitas. Em geral, se a cláusula contém l literais,

$$(a_1 \vee a_2 \vee \dots \vee a_l)$$

podemos substituí-la pelas $l - 2$ cláusulas

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z}_1 \vee a_3 \vee z_2) \wedge (\overline{z}_2 \vee a_4 \vee z_3) \wedge \dots \wedge (\overline{z}_{l-3} \vee a_{l-1} \vee a_l).$$

Podemos facilmente verificar que a nova fórmula é satisfável se e somente se a fórmula original o era, portanto a prova está completa.

7.5 Problemas NP-completos adicionais

Nesta seção apresentamos teoremas adicionais mostrando que várias linguagens são NP-completas. Nossa estratégia geral é exibir uma redução em tempo polinomial a partir de 3SAT para a linguagem em questão, embora às vezes reduzimos a partir de outras linguagens NP-completas quando isso é mais conveniente.

Quando construímos uma redução em tempo polinomial de 3SAT para uma linguagem, procuramos por estruturas naquela linguagem que possam simular as variáveis e cláusulas nas fórmulas booleanas. Tais estruturas são às vezes chamadas *dispositivos*. Por exemplo, na redução de 3SAT para CLIQUE apresentada no Teorema 7.26, os nós individualmente simulam variáveis e triplas de nós simulam cláusulas. Um nó individualmente pode ou não ser um membro do clique, o que corresponde a uma variável que pode ou ser verdadeira na atribuição que satisfaz a fórmula. Cada cláusula tem que conter um literal que tem o valor-verdade VERDADEIRO e que corresponde à forma

pela qual cada tripla tem que conter um nó no clique se o tamanho-alvo é para ser atingido. O corolário do Teorema 7.26 a seguir enuncia que *CLIQUE* é NP-completa.

Corolário 7.33
CLIQUE é NP-completa.

O problema da cobertura de vértices

Se G é um grafo não-direcionado, uma *cobertura de vértices* de G é um subconjunto dos nós onde toda aresta de G toca um daqueles nós. O problema da cobertura de vértices pergunta pelo tamanho da menor cobertura de vértices. Seja

$$\text{COBERT-VERT} = \{\langle G, k \rangle \mid G \text{ é um grafo não-direcionado que tem uma cobertura de vértices de } k\text{-nós}\}.$$

Teorema 7.34
 COBERT-VERT é NP-completa.

Prova. Damos uma redução de $3SAT$ para COBERT-VERT que opera em tempo polinomial. A redução mapeia uma fórmula booleana ϕ num grafo G e um valor k . Cada aresta em G tem que tocar em pelo menos um nó na cobertura de vértices, de modo que um dispositivo natural para uma variável é uma única aresta. Fazendo aquela variável ser VERDADEIRO corresponde a selecionar o nó esquerdo para a cobertura de vértices, enquanto que fazendo-a ser FALSO corresponde ao nó direito. Rotulamos os dois nós no dispositivo para a variável x como x e \bar{x} .

Os dispositivos para as cláusulas são um pouco mais complexos. Cada dispositivo de cláusula é uma tripla de três nós que são rotulados com os três literais da cláusula. Esses três nós são conectados uns aos outros e aos nós nas variáveis-dispositivos que têm rótulos idênticos. Por conseguinte o número total de nós que aparecem em G é $2m + 3l$, onde ϕ tem m variáveis e l cláusulas. Suponha que k seja $m + 2l$.

Por exemplo, se $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$, a redução produz $\langle G, k \rangle$ a partir de ϕ , onde $k = 8$ e G toma a forma mostrada na Figura 7.11.

Figura 7.11: O grafo que a redução produz a partir de $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$

Para provar que essa redução funciona, precisamos mostrar que ϕ é satisfatível se e somente se G tem uma cobertura de vértices

O problema do caminho hamiltoniano

Lembre-se que o problema do caminho hamiltoniano pergunta se o grafo de entrada contém um caminho de s para t que passa por todo nó exatamente uma vez.

Teorema 7.35

CAM-HAMIL é NP-completo.

Idéia da prova. Para mostrar que *CAM-HAMIL* é NP-completo temos que demonstrar duas coisas: (1) que *CAM-HAMIL* está em NP; e (2) que toda linguagem A em NP é reduzível em tempo polinomial a *CAM-HAMIL*. O primeiro fizemos na Seção 7.3. Para fazer o segundo mostramos que um problema NP-completo conhecido, *3SAT*, é reduzível em tempo polinomial a *CAM-HAMIL*. Damos uma maneira de converter 3fnc-fórmulas em grafos nos quais caminhos hamiltonianos correspondem a atribuições que satisfazem a fórmula. Os grafos contêm dispositivos que imitam variáveis e cláusulas. O dispositivo de variáveis é uma estrutura em forma de losango que pode ser percorrida de duas maneiras, correspondendo às duas atribuições de valor-verdade. O dispositivo de cláusulas é um nó. Assegurar que o caminho passa por cada dispositivo de cláusula corresponde a assegurar que cada cláusula é satisfeita na atribuição que satisfaz a fórmula.

Prova. Anteriormente mostramos que *CAM-HAMIL* está em NP, portanto tudo o que resta ser feito é mostrar que $3SAT \leq_P CAM-HAMIL$. Para cada 3fnc-fórmula ϕ mostramos como construir um grafo direcionado G com dois nós, s e t , onde um caminho hamiltoniano existe entre s e t se ϕ é satisfatível.

Começamos a construção com uma 3cnf-fórmula ϕ contendo k cláusulas,

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

onde cada a , b , e c é um literal x_i ou \bar{x}_i . Sejam x_1, \dots, x_l as l variáveis de ϕ .

Agora mostramos como converter ϕ em um grafo G . O grafo G que construímos tem várias partes para representar as estruturas (variáveis e cláusulas) que aparecem em ϕ .

Representamos cada variável x_i com uma estrutura em formato de losango que contém uma linha horizontal de nós, como mostrado na Figura 7.12. Mais adiante especificamos o número de nós que aparecem na linha horizontal.

Figura 7.12: Representando a variável x_i como uma estrutura em forma de losango

Representamos cada cláusula de ϕ como um único nó, da seguinte maneira:

Figura 7.13: Representando a cláusula c_j como um nó

A Figura 7.14 mostra a estrutura global de G . Ela mostra todos os elementos de G e seus relacionamentos, exceto as arestas que representam o relacionamento das variáveis às cláusulas que as contêm.

Figura 7.14: A estrutura de alto-nível de G

A seguir mostramos como conectar os losangos representando as variáveis aos nós representando as cláusulas. Cada estrutura em forma de losango contém uma linha horizontal de nós conectados por arestas indo em ambas as direções. A linha horizontal contém $3k + 1$ nós além dos dois nós nas extremidades pertencentes ao losango. Esses nós são agrupados em pares adjacentes, um para cada cláusula, com nós separadores extra em seguida aos pares, como mostrado na Figura 7.15.

Figura 7.15: Os nós horizontais em uma estrutura em forma de losango

Se a variável x_i aparece na cláusula c_j , adicionamos as duas arestas seguintes do j -ésimo par no i -ésimo losango para o j -ésimo nó cláusula.

Figura 7.16: As arestas adicionais quando a cláusula c_j contém x_i

Se \bar{x}_i aparece na cláusula c_j , adicionamos as duas arestas seguintes do j -ésimo par no i -ésimo losango para o j -ésimo nó cláusula.

Depois que adicionamos todas as arestas correspondentes a cada ocorrência de x_i ou \bar{x}_i em cada cláusula, a construção de G está completa. Para mostrar que essa construção funciona, argumentamos que, se ϕ é satisfatível, um caminho hamiltoniano existe de s para t e, reciprocamente, se tal caminho existe, ϕ é satisfatível.

Suponha que ϕ seja satisfatível. Para exibir um caminho hamiltoniano de s para t , primeiro ignoramos os nós-cláusula. O caminho começa em s , passa por cada losango a cada vez, e termina em t . Para atingir os nós horizontais em um losango, o caminho ou faz um zigue-zague da esquerda para a direita ou um zigue-zague através do losango correspondente. Se x_i recebe o valor VERDADEIRO, faça um zigue-zague através do losango correspondente. Se x_i recebe o valor FALSO, faça um zigue-zague. Mostramos ambas as possibilidades na Figura 7.18.

Até agora esse caminho cobre todos os nós em G exceto os nós cláusula. Podemos facilmente incluí-los adicionando desvios nos nós horizontais. Em cada cláusula, selecione um dos literais que recebem o valor VERDADEIRO pela atribuição que satisfaz a fórmula.

Se selecionamos x_i na cláusula c_j , podemos desviar no j -ésimo par no i -ésimo losango. Fazer isso é possível porque x_i tem que ser VERDADEIRO, portanto o caminho faz um zigue-zague da esquerda para a direita através do losango correspondente. Daí as arestas para o nó c_j estão na ordem correta para permitir um desvio e um retorno.

Figura 7.17: As arestas adicionais quando a cláusula c_j contém $\overline{x_i}$

Figura 7.18: Fazendo zigue-zague e zigue-zagueando através de um losango, conforme determinado pela atribuição que satisfaz

Igualmente, se selecionamos $\overline{x_i}$ na cláusula c_j , podemos desviar no j -ésimo par no i -ésimo losango. Fazer isso é possível porque x_i tem que ser FALSO, portanto o caminho zigue-zagueia da direita para a esquerda através do losango correspondente. Daí as arestas para o nó c_j estão novamente na ordem correta para permitir um desvio e um retorno. (Note que cada literal verdadeiro em uma cláusula fornece uma *opção* de um desvio para atingir o nó cláusula. Como um resultado, se diversos literais em uma cláusula são verdadeiros, somente um desvio é tomado.) Por conseguinte construímos o caminho hamiltoniano desejado.

Para a direção contrária, se G tem um caminho hamiltoniano de s para t , exibimos uma atribuição que satisfaz ϕ . Se o caminho hamiltoniano é *normal*, ou seja, passa pelos losangos na ordem e primeiro o de cima depois o de baixo, exceto para os desvios para os nós cláusula, podemos facilmente obter a atribuição que satisfaz a fórmula. Se o caminho zigue-zagueia através do losango, atribuímos à variável correspondente o valor VERDADEIRO, e, se o caminho faz um zigue-zague, atribuímos FALSO. Devido ao fato de que cada nó cláusula aparece no caminho, por meio da observação do losango no qual o desvio para ele é tomado, podemos determinar qual dos literais na cláusula é VERDADEIRO.

Tudo o que resta ser feito é mostrar que um caminho hamiltoniano tem que ser normal. A única maneira da normalidade falhar seria para o caminho entrar numa cláusula a partir de um losango mas retornar para um outro, como na Figura 7.19. O caminho vai do nó a_1 para c , mas ao invés de retornar a a_2 ou a_3 tem que ser um nó separador. Se a_2 fosse um nó separador, as únicas arestas entrando em a_2 seriam de a_1 e a_3 . Se a_3 fosse um nó separador, a_1 e a_2 estariam no mesmo par de cláusulas, e portanto as únicas arestas entrando em a_2 seriam de a_1 , a_3 , e c . Em qualquer caso, o caminho não poderia conter o nó a_2 . O caminho não pode entrar em a_2 a partir de a_3 , porque a_3 é o único nó disponível para o qual a_2 aponta, portanto o caminho tem que sair de a_2 via a_3 . Portanto um caminho hamiltoniano tem que ser normal. Essa redução obviamente opera em tempo polinomial e a prova está completa.

Figura 7.19: Essa situação não pode ocorrer

A seguir consideraremos uma versão não-direcionada do problema do caminho hamiltoniano, chamado *CAM-HAMIL-ND*. Para mostrar que *CAM-HAMIL-ND* é NP-completo damos uma redução em tempo polinomial a partir da versão direcionada do problema.

Teorema 7.36
CAM-HAMIL-ND é NP-completo.

Prova. A redução toma um grafo direcionado G com nós s e t , e constrói um grafo não-direcionado G' com nós s' e t' . O grafo G tem um caminho hamiltoniano de s para t se e somente se G' tem um caminho hamiltoniano de s' para t' . Descrevemos G' da seguinte maneira.

Cada nó u de G , exceto s e t , é substituído por uma tripla de nós u^{entra} , u^{meio} e u^{sai} em G' . Os nós s e t em G são substituídos pelos nós s^{sai} e t^{entra} em G' . Areias de dois tipos aparecem em G' . Primeiro, arestas conectam u^{meio} com u^{entra} e u^{sai} . Segundo, uma aresta conecta u^{sai} com v^{entra} se uma aresta vai de u para v em G . Isso completa a construção de G' .

Podemos demonstrar que essa construção funciona mostrando que G tem um caminho hamiltoniano de s para t se e somente se G' tem um caminho hamiltoniano de s^{sai} para t^{entra} . Para mostrar uma direção, observamos que um caminho hamiltoniano P em G ,

$$s, u_1, u_2, \dots, u_k, t,$$

tem um caminho hamiltoniano correspondente P' em G' ,

$$s^{\text{sai}}, u_1^{\text{entra}}, u_1^{\text{meio}}, u_1^{\text{sai}}, u_2^{\text{entra}}, u_2^{\text{meio}}, u_2^{\text{sai}}, \dots, t^{\text{entra}}.$$

Para mostrar a outra direção, afirmamos que qualquer caminho hamiltoniano em G' de s^{sai} para t^{entra} em G' tem que ir de uma tripla de nós para uma tripla de nós, exceto o início e o fim, como faz o caminho P' que acabamos de descrever. Isso completaria a prova porque qualquer caminho desse tem um caminho hamiltoniano correspondente em G . Provamos a afirmação seguindo o caminho começando no nó s^{sai} . Observe que o nó seguinte no caminho tem que ser u_i^{entra} para algum i porque somente aqueles nós estão conectados a s^{sai} . O nó seguinte tem que ser u_i^{meio} , porque nenhuma outra maneira está disponível para incluir u_i^{meio} no caminho hamiltoniano. Após u_i^{meio} vem u_j^{sai} porque esse é o único outro nó ao qual u_i^{meio} está conectado. O nó seguinte tem que ser u_j^{entra} para algum j porque nenhum outro nó disponível está conectado a u_j^{sai} . O argumento então repete até que t^{sai} seja atingido.

O problema da soma de subconjuntos

Retomemos o problema *SOMA-SUBCONJ* definido na página 246. Naquele problema, nos é dada uma coleção de números, x_1, \dots, x_k juntamente com um número alvo t , e deseja-se determinar se a coleção contém uma subcoleção cuja soma é t . Agora mostramos que esse problema é NP-completo.

Teorema 7.37
SOMA-SUBCONJ é NP-completo.

Idéia da prova. Já mostramos que *SOMA-SUBCONJ* está em NP no Teorema 7.21. Provamos que todas as linguagens em NP são redutíveis em tempo polinomial a *SOMA-SUBCONJ* reduzindo a linguagem NP-completa *3SAT* a ele. Dada uma 3fnc-fórmula ϕ construímos uma instância do problema *SOMA-SUBCONJ* que contém uma subcoleção cuja soma é t se e somente se ϕ é satisfatível. Chame essa subcoleção T .

Para obter essa redução encontramos estruturas do problema *SOMA-SUBCONJ* que representam variáveis e cláusulas. A instância do problema *SOMA-SUBCONJ*

que construimos contém números de grande magnitude apresentados em notação decimal. Representamos variáveis por pares de números e cláusulas por certas posições nas representações decimais dos números.

Representamos a variável x_i por dois números, y_i e z_i . Provamos que ou y_i ou z_i tem que estar em T para cada i , o que estabelece a codificação para o valor-verdade de x_i na atribuição que satisfaz a fórmula.

Cada posição na cláusula contém um certo valor no alvo t , que impõe um requisito sobre o subconjunto T . Provamos que esse requisito é o mesmo que aquele sobre a cláusula correspondente, a saber, que um dos literais naquela cláusula seja atribuído o valor VERDADEIRO.

Prova. Já sabemos que $SOMA-SUBCONJ \in NP$, portanto agora mostramos que $3SAT \leq_P SOMA-SUBCONJ$.

Seja ϕ uma fórmula booleana com variáveis x_1, \dots, x_l e cláusulas c_1, \dots, c_k . A redução converte ϕ para uma instância do problema $SOMA-SUBCONJ \langle S, t \rangle$, na qual os elementos de S e o número t são as linhas na seguinte tabela expressa em notação decimal comum. As linhas acima da linha dupla são rotuladas

$$y_1, z_1, y_2, z_2, \dots, y_l, z_l \quad \text{e} \quad g_1, h_1, g_2, h_2, \dots, g_k, h_k$$

e compreendem os elementos de S . A linha abaixo da linha dupla é a linha t .

Por conseguinte S contém um par de números y_i, z_i , para cada variável x_i em ϕ . A representação decimal desses números está em duas partes, como indicado na tabela. A parte da esquerda compreende um 1 seguido de $l - i$ 0's. A parte da direita contém um dígito para cada cláusula, onde o j -ésimo dígito de y_i é 1 se a cláusula c_j contém o literal x_i e o j -ésimo dígito de z_i é 1 se a cláusula c_j contém o literal $\overline{x_i}$. Dígitos não especificados como sendo 1 são 0.

A tabela é parcialmente preenchida para ilustrar cláusulas-amostra, c_1, c_2 , e c_k :

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \dots) \wedge \dots \wedge (\overline{x_3} \vee \dots \vee \dots).$$

Adicionalmente, S contém um par de números, g_j, h_j , para cada cláusula c_j . Esses dois números são iguais e consistem de um 1 seguido de $k - j$ 0's.

Finalmente, o número alvo t , a última linha da tabela, consiste de l 1's seguidos de k 3's.

Agora mostramos por que essa construção funciona. Demonstramos que ϕ é satisfatível se e somente se algum subconjunto S tem soma igual a t .

Suponha que ϕ seja satisfatível. Construimos um subconjunto de S da seguinte maneira. Selecionemos y_i se x_i recebe o valor VERDADEIRO na atribuição que satisfaz ϕ e z_i se x_i recebe o valor FALSO. Se somarmos o que selecionamos até então, obtemos um 1 em cada um dos l primeiros dígitos porque selecionamos ou y_i ou z_i para cada i . Além do mais, cada um dos últimos k dígitos é um número entre 1 e 3 porque cada cláusula é satisfeita e portanto contém entre 1 e 3 literais verdadeiros. Agora selecionamos mais números g e h em quantidade suficiente para trazer cada um dos últimos k dígitos para 3, e portanto atingindo o alvo.

Suponha que um subconjunto de S tem soma igual a t . Construimos uma atribuição que satisfaz ϕ após fazer várias observações. Primeiro, todos os dígitos nos membros de S são 0 ou 1. Além do mais, cada coluna na tabela que descreve S contém no

máximo cinco 1's. Logo um “transporte” para a próxima coluna nunca ocorre quando um subconjunto de S é somado. Para obter um 1 em cada uma das l primeiras colunas o subconjunto tem que ter ou y_i ou z_i para cada i , mas não ambos.

Agora montamos uma atribuição que satisfaz ϕ . Se o subconjunto contém y_i , atribuimos VERDADEIRO a x_i , caso contrário, atribuimos FALSO. Essa atribuição tem que satisfazer ϕ porque em cada uma das k colunas finais a soma é sempre 3. Na coluna c_j , no máximo 2 podem vir de g_j e h_j , portanto pelo menos 1 nessa coluna tem que vir de algum y_i ou z_i no subconjunto. Se for y_i , então x_i aparece em c_j e recebe VERDADEIRO, portanto c_j é satisfeita. Se for z_i , então \bar{x}_i aparece em c_j e x_i recebe FALSO, portanto c_j é satisfeita. Por conseguinte ϕ é satisfeita.

Finalmente, temos que ter certeza de que a redução pode ser realizada em tempo polinomial. A tabela tem um tamanho de aproximadamente $(k + l)^2$, e cada entrada é facilmente calculada para qualquer ϕ . Portanto o tempo total é $O(n^2)$ estágios fáceis.

Exercícios

7.1 Para cada item abaixo diga se é VERDADEIRO ou FALSO.

- a. $2n = O(n)$.
- b. $n^2 = O(n)$.
- c. $n^2 = O(n \log^2 n)$.
- d. $n \log n = O(n^2)$.
- e. $3^n = 2^{O(n)}$.
- f. $2^{2^n} = O(2^{2^n})$.

7.2 Para cada item abaixo diga se é VERDADEIRO ou FALSO.

- a. $n = o(2n)$.
- b. $2n = o(n^2)$.
- c. $2^n = o(3^n)$.
- d. $1 = o(n)$.
- e. $n = o(\log n)$.
- f. $1 = o(1/n)$.

7.3 Quais dos seguintes pares de números são primos entre si? Mostre os cálculos que levaram a suas conclusões.

- a. 1274 e 10505
- b. 7289 e 8029

7.4 Preencha a tabela descrita no algoritmo de tempo polinomial para reconhecimento de linguagem livre-do-contexto do Teorema 7.14 para a cadeia $w = \text{baba}$ e a GLC G :

$$\begin{array}{rcl} S & \rightarrow & RT \\ R & \rightarrow & TR \mid a \\ T & \rightarrow & TR \mid b \end{array}$$

7.5 A fórmula abaixo é satisfatível?

$$(x \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee y) \wedge (\bar{x} \vee \bar{y})$$

7.6 Mostre que P é fechada sob união, concatenação, e complemento.

7.7 Mostre que NP é fechada sob união e concatenação.

7.8 Mostre que testar primalidade é solúvel em tempo polinomial se usarmos uma codificação unária ao invés de uma codificação binária para números. Em outras palavras, mostre que a linguagem $PRIMOS-UNARIOS = \{1^n \mid n \text{ é primo}\}$ está em P.

7.9 Seja $CONEXO = \{\langle G \rangle \mid G \text{ é um grafo não-direcionado conexo}\}$. Analise o algoritmo dado na página 145 para mostrar que essa linguagem está em P.

7.10 Um *triângulo* em um grafo não-direcionado é um 3-clique. Mostre que $TRIANGULO \in P$, onde $TRIANGULO = \{\langle G \rangle \mid G \text{ contém um triângulo}\}$.

7.11 Chame os grafos G e H *isomorfos* se os nós de G podem ser reordenados de modo que ele fique idêntico a H . Seja $ISO = \{\langle G, H \rangle \mid G$ e H são grafos isomorfos $\}$. Mostre que $ISO \in NP$.

Problemas

7.12 Seja

$$\begin{aligned} EXPMOD &= \{\langle a, b, c, p \rangle \mid a, b, c, \text{ e } p \text{ são inteiros em binário} \\ &\quad \text{tais que } a^b \equiv c \pmod{p}\}. \end{aligned}$$

Mostre que $EXPROMOD \in P$. (Note que o algoritmo mais óbvio não roda em tempo polinomial. Dica: Tente primeiro onde b é uma potência de 2.)

7.13 Mostre que P é fechada sob a operação estrela. (Dica: Sobre a entrada $y = y_1 \cdots y_n$ para $y_i \in \Sigma$, construa uma tabela indicando para cada $i \leq j$ se a subcadeia $y_i \cdots y_j \in A^*$ para qualquer $A \in P$.)

7.14 Mostre que NP é fechada sob a operação estrela.

7.15 Seja $SSOMA-UNARIO$ o problema da soma de subconjuntos no qual todos os números são representados em unário. Por que a prova de NP-completude para $SUMA-SUBCONJ$ falha em mostrar que $SSOMA-UNARIO$ é NP-completa? Mostre que $SSOMA-UNARIO \in P$.

7.16 Suponha G que represente um grafo não-direcionado e seja

$$\begin{aligned} CAMINHC &= \{\langle G, a, b, k \rangle \mid G \text{ contém um caminho simples de} \\ &\quad \text{comprimento no máximo } k \text{ de } a \text{ para } b\}. \end{aligned}$$

e

$$\begin{aligned} CAMINHL &= \{\langle G, a, b, k \rangle \mid G \text{ contém um caminho simples de} \\ &\quad \text{comprimento no mínimo } k \text{ de } a \text{ para } b\}. \end{aligned}$$

a. Mostre que $CAMINHC \in P$.

- b. Mostre que *CAMINHL* é NP-completa. Você pode assumir que NP-completude de *CAMHAMILNAOD*, o problema do caminho hamiltoniano para grafos não-direcionado.

7.17 Mostre que, se P=NP então toda linguagem $A \in P$ exceto se $A = \emptyset$ e $A = \Sigma^*$ é NP-completa.

***7.18** Mostre que $PRIMOS = \{n \mid n \text{ é um número primo em binário}\} \in NP$. (Dica: Para $p > 1$ o grupo multiplicativo $Z_p^* = \{x \mid x \text{ é primo em relação a } p \text{ e } 1 \leq x < p\}$ é tanto cíclico quanto de ordem $p - 1$ sse p é primo. Você pode usar esse fato sem prová-lo.)

7.19 Seja $DUPLO-SAT = \{\langle \phi \rangle \mid \phi \text{ tem pelo menos duas valorações que a satisfazem}\}$. Mostre que *DUPLO-SAT* é NP-completa.

7.20 Uma **permutação** sobre o conjunto $\{1, \dots, k\}$ é uma função sobrejetora um-para-um sobre esse conjunto. Quando p é uma permutação, p^t significa a composição de p com ela própria t vezes. Seja

$$\begin{aligned} POTENC-PERM = \{&\langle p, q, t \rangle \mid p = q^t \text{ onde } p \text{ e } q \text{ são permutações} \\ &\text{sobre } \{1, \dots, k\} \text{ e } t \text{ é um inteiro em binário}\}. \end{aligned}$$

Mostre que *POTENC-PERM* $\in P$. (Note que o algoritmo mais óbvio não roda dentro de tempo polinomial. Dica: Primeiro experimente-o onde t é uma potência de 2).

7.21 Seja

$MEIO-CLIQUE = \{\langle G \rangle \mid G \text{ é um grafo não-direcionado tendo um subgrafo completo}$

com pelo menos $n/2$ nós, onde n é o número de nós em $G\}$.

Mostre que *MEIO-CLIQUE* é NP-completa.

7.22 Seja ϕ uma 3fnc-fórmula. Uma **\neq -atribuição** às variáveis de ϕ é uma na qual cada cláusula contém dois literais com valores-verdade diferentes. Em outras palavras uma \neq -atribuição satisfaz ϕ sem atribuir verdadeiro a três literais em nenhuma cláusula.

- Mostre que a negação de qualquer \neq -atribuição a ϕ é também uma \neq -atribuição.
- Seja $\neq SAT$ a coleção de 3fnc-fórmulas que têm uma \neq -atribuição. Mostre que obtemos uma redução em tempo polinomial de $3SAT$ para $\neq SAT$ substituindo cada cláusula c_i

$$(y_1 \vee y_2 \vee y_3)$$

pelas duas cláusulas

$$(y_1 \vee y_2 \vee z_i) \text{ e } (\overline{z}_i \vee y_3 \vee b)$$

onde z_i é uma nova variável para cada cláusula c_i e b é uma única variável adicional.

- Conclua que $\neq SAT$ é NP-completa.

- 7.23** Um *corte* em um grafo não-direcionado é uma separação dos vértices V em dois subconjuntos disjuntos S e T . O tamanho de um corte é o número de arestas que têm uma extremidade em S e a outra em T . Seja

$$\text{CORTE-MAX} = \{\langle G, k \rangle \mid G \text{ é um corte de tamanho } k \text{ ou mais}\}.$$

Mostre que *CORTE-MAX* é NP-completa. Você pode assumir o resultado do Problema 7.22. (Dica: Mostre que $\neq \text{SAT} \leq_P \text{CORTE-MAX}$.

Capítulo 8

Complexidade de Espaço

Neste capítulo consideraremos a complexidade de problemas computacionais em termos da quantidade de espaço (ou memória) que eles requerem. Tempo e espaço são duas das mais importantes considerações quando buscamos soluções práticas para muitos problemas computacionais. Complexidade de espaço compartilha muitas das características da complexidade de tempo e serve como uma forma adicional de classificação de problemas conforme sua dificuldade computacional.

Como fizemos com complexidade de tempo, precisamos selecionar um modelo para medir o espaço usado por um algoritmo. Continuamos com o modelo da máquina de Turing pela mesma razão que o utilizamos para medir tempo. Máquinas de Turing são matematicamente simples e suficientemente próximas a computadores reais para dar resultados significativos.

Definição 8.1

Seja M uma máquina de Turing determinística que pára sobre todas as entradas. Definimos a **complexidade de espaço** de M como sendo a função $f : \mathcal{N} \rightarrow \mathcal{N}$, onde $f(n)$ é o número máximo de células da fita que M varre sobre qualquer entrada de comprimento n . Se a complexidade de espaço de M é $f(n)$, também dizemos que M roda em espaço $f(n)$.

Se M é uma máquina de Turing não-determinística na qual todos os ramos páram sobre todas as entradas, definimos sua complexidade de espaço $f(n)$ como sendo o número máximo de células de fita que M varre sobre qualquer ramo de sua computação para qualquer entrada de comprimento n .

Como no caso da complexidade de tempo, usualmente estimamos a complexidade de espaço de máquinas de Turing usando notação assintótica.

Definição 8.2

Seja $f : \mathcal{N} \rightarrow \mathcal{N}$ uma função. As **classes de complexidade de espaço**, $\text{SPACE}(f(n))$ e $\text{NSPACE}(f(n))$, são definidas da seguinte forma:

$$\text{SPACE}(f(n)) = \{L \mid L \text{ é uma linguagem decidida por uma máquina de Turing determinística de complexidade de espaço } O(f(n))\}.$$

$$\text{NSPACE}(f(n)) = \{L \mid L \text{ é uma linguagem decidida por uma máquina de Turing}$$

não-determinística de complexidade de espaço $O(f(n))\}.$

Exemplo 8.3

No Capítulo 7 introduzimos o problema NP-completo *SAT*. Aqui, mostramos que *SAT* pode ser resolvido com um algoritmo de espaço linear. Acreditamos que *SAT* não possa ser resolvido com um algoritmo de tempo polinomial, muito menos com um algoritmo de tempo linear, porque *SAT* é NP-completo. Espaço parece ser mais poderoso que tempo porque espaço pode ser reusado, enquanto que tempo não pode.

M_1 = “Sobre a entrada $\langle\phi\rangle$, onde ϕ é uma fórmula booleana:

1. Para cada atribuição de verdade às variáveis x_1, \dots, x_m de ϕ :
2. Calcule o valor de ϕ para essa atribuição.
3. Se ϕ chegou a ter valor 1, *aceite*; se não, *rejeite*.

A máquina M_1 claramente roda em espaço linear porque cada iteração do laço pode reusar a mesma porção da fita. A máquina somente necessita de armazenar a atribuição de verdade corrente e isso pode ser feito em espaço $O(m)$. O número de variáveis m é no máximo n , o comprimento da entrada, portanto essa máquina roda em espaço $O(n)$.

Exemplo 8.4

Aqui, ilustramos a complexidade de espaço não-determinístico de uma linguagem. Na próxima seção mostramos como determinar a complexidade de espaço não-determinístico pode ser útil para determinar sua complexidade de espaço determinístico. Considere o problema de se testar se um autômato finito não-determinístico

8.1 Teorema de Savitch

8.2 A classe PSPACE

8.3 PSPACE-completude

8.4 As classes L e NL

8.5 NL-completude

8.6 NL é igual a coNL

Capítulo 9

Intratabilidade

Certos problemas computacionais são solúveis em princípio, mas as soluções requerem tanto tempo ou espaço que eles não podem ser usados na prática. Tais problemas são chamados *intratáveis*. Intratabilidade pode tomar quaisquer formas, dependendo dos recursos computacionais disponíveis e o tipo de solução desejada. Por exemplo, um problema que é fácil resolver a maior parte do tempo, ainda assim ocasionalmente difícil, seria intratável somente no pior caso. Ou um problema pode ser facilmente solúvel em um supercomputador mas pode requerer uma quantidade desordenada de tempo em um computador pessoal. Focamos nos problemas cuja complexidade no pior caso é tão enorme que qualquer computador que poderíamos possivelmente construir necessaria de mais tempo do que se acredita que resta no tempo de vida do universo.

Nos Capítulos 7 e 8, introduzimos diversos problemas considerados como sendo intratáveis mas nenhum que seja demonstravelmente intratável. Por exemplo, a maioria das pessoas acreditam que o problema *SAT* e todos os outros problemas NP-completos sejam intratáveis, embora não saibamos como provar que eles o são. Neste capítulo damos exemplos de problemas que podemos provar serem intratáveis.

De modo a apresentar esses exemplos, desenvolvemos vários teoremas que relacionam o poder de máquinas de Turing à quantidade de tempo ou espaço disponível para computação. Concluimos o capítulo com uma discussão da possibilidade de se provar que problemas em NP são intratáveis e dessa forma se resolver a questão P versus NP. Primeiro, introduzimos a técnica da relativização e a usamos para argumentar que certos métodos não nos permitirão atingir esse objetivo. Aií então, discutimos a teoria da complexidade de circuitos, uma abordagem tomada por pesquisadores que tem mostrado ser promissora.

9.1 Teoremas de hierarquia

Senso comum sugere que dar a uma máquina de Turing mais tempo ou mais espaço deveria aumentar a classe de problemas que ela pode resolver. Em outras palavras, máquinas de Turing deveriam ser capazes de decidir mais linguagens em tempo n^3 do que elas podem em tempo n^2 . Essa intuição está correta, sujeita a certas condições. Os *teoremas de hierarquia* formalizam e provam essa intuição. Usamos o termo *teorema de hierarquia* porque cada um desses teoremas prova que as classes de complexidade

de tempo e de espaço não são todas iguais—elas formam uma hierarquia por meio da qual as classes com limitantes maiores contêm mais linguagens do que as classes com limitantes menores.

Primeiro apresentamos o teorema de hierarquia para complexidade de espaço porque ela é um pouco mais simples do que aquela para complexidade de tempo. Antes de chegar ao real enunciado do teorema, precisamos fazer a seguinte definição técnica.

Definição 9.1

Uma função $f : \mathcal{N} \rightarrow \mathcal{N}$, onde $f(n) \geq \log n$, é chamada **construtível em espaço** se a função que mapeia 1^n para a representação binária de $f(n)$ é computável em espaço $O(f(n))$.¹

Em outras palavras, f é construtível em espaço se alguma MT M existe que roda em espaço $O(f(n))$ e sempre pára com a representação binária de $f(n)$ sobre sua fita quando iniciada com a entrada 1^n . Funções fracionárias tais como $n \log n$ e \sqrt{n} são aproximadas para o próximo inteiro menor para os propósitos de construtibilidade em tempo e espaço.

Exemplo 9.2

Todas as funções ocorrendo comumente que são pelo menos $\log n$ são construtíveis em espaço, incluindo as funções $\log n$, $n \log n$, e n^2 .

Por exemplo, n^2 é construtível em espaço porque uma máquina pode tomar sua entrada 1^n , obter n em binário contando o número de 1's, e dar como saída n^2 usando qualquer método padrão para multiplicar n por si próprio. O espaço total usado é $O(n)$, que é certamente $O(n^2)$.

Ao mostrar que funções $f(n)$ que são $o(n)$ são construtíveis em espaço, usamos uma fita de entrada somente-leitura separada, como fizemos quando definimos complexidade de espaço sublinear na Seção 8.4. Por exemplo, tal máquina pode computar a função que mapeia 1^n para a representação binária de $\log n$ da seguinte maneira. Ela primeiro conta o número de 1's na sua entrada em binário usando sua fita de trabalho à medida que ela move sua cabeça ao longo da fita de entrada. Aí então, com n em binário sobre sua fita de trabalho, ela pode computar $\log n$ contando o número de bits na representação binária de n .

O papel da construtibilidade de espaço na hierarquia de espaço pode ser entendido a partir da seguinte situação. Se $f(n)$ e $g(n)$ são dois limitantes de espaço, onde $f(n)$ é assintoticamente maior que $g(n)$, esperaríamos que uma máquina fosse capaz de computar mais linguagens em espaço $f(n)$ do que em espaço $g(n)$. Entretanto, suponha que $f(n)$ excede $g(n)$ por somente uma quantidade muito pequena e difícil de computar. Então, a máquina não pode ser capaz de usar o espaço extra proveitosamente porque até computar a quantidade de espaço extra pode requerer mais espaço do que está disponível. Nesse caso, uma máquina pode não ser capaz de computar mais linguagens em espaço $f(n)$ do que ela pode em espaço $g(n)$. Estipular que $f(n)$ é construtível em espaço evita essa situação e nos permite provar que uma máquina pode computar mais do que ela seria capaz em qualquer limitante assintoticamente menor, como o teorema seguinte mostra.

Teorema 9.3

¹Lembre-se que 1^n significa uma cadeia de n 1's.

Teorema da Hierarquia de Espaço Para qualquer função construtível em espaço $f : \mathcal{N} \rightarrow \mathcal{N}$, existe uma linguagem A que é decidível em espaço $O(f(n))$ mas não em espaço $o(f(n))$.

.....
Idéia da prova. Temos que exibir uma linguagem A que tem duas propriedades. A primeira diz que A é decidível em espaço $O(f(n))$. A segunda diz que A não é decidível em espaço $o(f(n))$.

Descrevemos A dando um algoritmo B que a decide. O algoritmo B rodará em espaço $O(f(n))$, dessa forma assegurando a primeira propriedade. Além do mais, B garantirá que A é diferente de qualquer linguagem que é decidível em espaço $o(f(n))$, dessa forma assegurando a segunda propriedade.

Não espere obter uma imagem mental tão simples e clara da linguagem A quanto você pode ter para as outras linguagens que apareceram até agora neste livro. Essas outras linguagens têm sido descritas como coleções de cadeias satisfazendo propriedades especificadas. Dessa descrição semântica, temos frequentemente prosseguido para dar um algoritmo para testar pertinência na linguagem. A linguagem A é diferente no sentido de que ela é descrita somente por um algoritmo e lhe falta uma definição mais simples, não-algorítmica.

De modo a assegurar que A não seja decidível em espaço $o(f(n))$, desenhamos B para implementar o método da diagonalização que usamos para provar a insolubilidade do problema da parada A_{MT} no Teorema 4.9 da página 165. Se M é uma máquina que decide uma linguagem em espaço $o(f(n))$, B garante que A difere da linguagem de M em pelo menos um lugar. Que lugar? O lugar correspondente a uma descrição de si própria.

Vamos olhar para a maneira pela qual B opera. Grosso modo, B toma sua entrada como sendo a descrição de uma MT. (Se a entrada não for a descrição de nenhuma MT, então a ação de B é inconseqüente em relação a sua entrada, portanto arbitrariamente fazemos B rejeitar.) Se M pára dentro daquela quantidade de espaço, B aceita sse M rejeita. Se N não pára, B simplesmente rejeita. Portanto se M roda dentro de espaço $f(n)$, B tem espaço suficiente para assegurar que sua linguagem é diferente da de M . Senão, B não tem espaço suficiente para adivinhar o que M faz, mas felizmente B não tem qualquer requisito de agir diferentemente de máquinas que não rodam em espaço $o(f(n))$, portanto a ação de B sobre sua entrada é inconseqüente.

Essa descrição captura a essência da prova mas omite diversos detalhes importantes. Se M roda em espaço $o(f(n))$, B tem que garantir que sua linguagem é diferente da linguagem de M . Mas mesmo quando M roda em espaço $o(f(n))$, ela pode usar mais que espaço $f(n)$ para n pequeno, quando o comportamento assintótico ainda não “disparou”. Possivelmente, B pode não ter espaço suficiente para rodar M até o fim sobre a entrada $\langle M \rangle$, e portanto B perderá sua única oportunidade de evitar a linguagem de M . Logo, se não formos cuidadosos, B pode acabar decidindo a mesma linguagem que M decide, e o teorema não seria provado.

Podemos resolver esse problema modificando B para dar-lhe oportunidades adicionais de evitar a linguagem de M . Ao invés de rodar M somente quando B recebe a entrada $\langle M \rangle$, ele roda M sempre que ele recebe uma entrada da forma $\langle M \rangle 10^*$, ou seja, uma entrada da forma $\langle M \rangle$ seguida de um 1 e algum número de 0's. Aí então, se M realmente está rodando em espaço $o(f(n))$, B terá espaço suficiente para rodá-la até o fim sobre a entrada $\langle M \rangle 01^k$ para algum valor grande de k porque o comportamento assintótico tem que eventualmente disparar.

Uma última questão técnica surge. Quando B roda M sobre alguma cadeia, M

pode cair num laço infinito embora que usando somente uma quantidade finita de espaço. Mas B supostamente é um decisor portanto temos que assegurar que B não entra em loop enquanto está simulando M . Qualquer máquina que roda em espaço $o(f(n))$ usa somente tempo $2^{o(f(n))}$. Modificamos B de modo que ele conta o número de passos usados na simulação de M . Se esse contador em algum momento excede $2^{f(n)}$, então B rejeita.

9.2 Relativização

9.3 Complexidade de circuito

Capítulo 10

Tópicos Avançados em Teoria da Complexidade

10.1 Algoritmos de aproximação

10.2 Algoritmos probabilísticos

10.3 Alternação

10.4 Sistemas de prova interativa

10.5 Computação paralela

10.6 Criptografia

Bibliografia Selecionada

1. ADLEMAN, L. Two theorems on random polynomial time. In *Proceedings of the Nineteenth IEEE Symposium on Foundations of Computer Science* (1978), pp. 75–83.

Glossário de traduções

Tradução Termo original em inglês

Índice Remissivo

computador, 9